E-MODEL: EVENT-BASED GRAPH DATA MODEL THEORY AND IMPLEMENTATION

A Thesis
Presented to
The Academic Faculty

by

Pilho Kim

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Electrical and Computer Engineering

Georgia Institute of Technology
August 2009

ACKNOWLEDGEMENTS

This work could never have been completed without the support and encouragement of numerous people who have encouraged and supported me in my pursuit of the research necessary for this undertaking.

I am deeply grateful to my advisor, Dr. Vijay Madisetti. His guidance always has been clear, insightful, and succinct and kept me focused on my goal. I also thank my former advisor, Dr. Ramesh Jain, for penetrating discussions and advice that inspired my research on topics that reflected our mutual interests. His consistent endeavor to support me even while he was fighting cancer will remain with me forever as the ideal role model for a professor.

I am also deeply appreciative of Dr. Chin-Hui Lee, Dr. Sudhakar Yalamanchili, Dr. Nikil Jayant, and Dr. Umakishore Ramachandran for their service on my dissertation committee and for their time spent reviewing the drafts of this study and ensuring that I met their high standards. Their comments were invaluable in helping me refine and strengthen my work.

Finally, no words are adequate for the gratitude I feel toward my family for their love and support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

The necessity of managing disparate data models is increasing within all IT areas. Emerging hybrid relational-XML systems are under development in this context to support both relational and XML data models. Typical hybrid systems are based on one type of database system, with support added to accommodate the other data model. However, there are ever-growing needs for adequate data models for texts and multimedia, which are applications that require proper storage, and their capability to coexist and collaborate with other data models is as important as that of a relational-XML hybrid model.

The problem is then whether relational, XML, or current hybrid database systems are flexible enough to accommodate additional text and multimedia data models. Although raw data type of texts involve simple characters, their document structures and relations between texts are not simple at all. Multimedia poses rather simple relations between data objects, but spatio-temporal variations in content complicate efforts to deal with this type. When multimedia and texts are merged for multimedia information service, then the complexity of content management rapidly exceeds the capabilities of even hybrid databases.

This work proposes a new data model named E-model that supports rich relations and reflects the dynamic nature of information. This E-model introduces abstract data typing objects and rules of relation that support: (1) the notion of time in object definition and relation, (2) multiple-type relations, (3) complex schema modeling methods using a relational directed acyclic graph, and (4) interoperation with popular data models.

Implementing a database system for a new data model takes good efforts for evaluation. The E-model structures first proposed in theories are materialized as database storage objects. To free the E-model from the back-end system, object identities are encapsulated in a unified data type named an e-node. Relations between sets of e-nodes conform to the

temporal flow to capture information dynamics.

This work describes the complete data model design process from an abstract data modeling rule design to actual database system implementation. To implement the E-model prototype, extensive data operation APIs have been developed on top of relational databases. In processing dynamic queries, our prototype achieves an order of magnitude improvement in speed compared with popular data models. Based on extensive E-model APIs, a new language named EML is proposed. EML extends the SQL-89 standard with various E-model features: (1) unstructured queries, (2) unified object namespaces, (3) temporal queries, (4) ranking orders, (5) path queries, (6) semantic expansions, and (7) natural joins. The E-model system can interoperate with popular data models with its rich relations and flexible structure to support complex data models. It can act as a stand-alone database server or it can also provide materialized views for interoperation with other data models. It can also co-exist with established database systems as a centralized online archive or as a proxy database server.

The current E-model prototype system was implemented on top of a relational database. This allows significant benefits from established database engines in application development. However, because of E-model's graph-based architecture, a graph walk on an E-model needs a sequence of joins which relational database systems, do not support efficiently. Our next research topic thus will be to develop an E-model hybrid database storage engine optimized for E-model architecture.

CHAPTER I

INTRODUCTION

A data model is a set of formal definitions on entities, properties, relations and operations. It defines how objects are formulated and linked with each other. Relational, hierarchical, network, object-oriented models exemplify such cases that have been developed in the database field. A database system is the implementation of a data model that is focused on its serving as a storage system. Database systems exist in a variety of forms, and each database system has pros and cons that determine its application domains.

Recently the integration of multiple applications, each of which has its own heterogeneous data model, has become one of the most challenging topics of work in the IT field. This is because this integration problem relates directly to the cost of system development and maintenance. The interconnection between disparate data silos over distributed database systems needs a generalized abstract data model that can interpret the formalisms and relations of heterogeneous data models. Such a new data model should be able to accommodate the dynamics of data structures. By dynamics, we mean the evolution of the data structure over time. Integration of data objects from different disciplines also causes semantic interpretation problems due to the differences in ontological object identities, their properties, and the types of relations between objects. This work pursues development of a new data model to meet such needs.

The developmental process from design to implementation for a new data model is long and arduous . Many proprietary data models have been born and disappeared, often without the implementation necessary for their validation. Also most literature on databases treats design and implementation issues as separate fields.

For instance in the field of relational databases, relational data model design, and relational database system (RDBMS) implementation are different research subjects. The work explored here, however, is not limited to abstract data model design but extends into implementation to present an enhanced database system.

The first chapter provides an overview of the data model design. Section 1.1describes the motivation that led us to undertake the design and implementation of a new data model. Section 1.2 discusses challenging problems from related fields. And Section 1.3 introduces the general data model design process that this work followed up.

1.1 Motivation

The evolution of information technology with its wealth of support for multimodal interaction is encouraging people to develop ambitious systems. Such progress accompanies developments in hardware. However, disparate data sources emerging from new technologies raise concerns about data complexity, their heuristic nature and the ever-growing costs to integrate and maintain this disparate data. Notable problems are classified into three categories: (1) Impedance mismatches in fundamental data objects and their properties; (2) disparities in necessary relation types and the methods to build relations between objects; and (3) difficulties in building a common interoperation protocol, within disparate data models. A mixture of these problems currently occur in most data models and are portrayed using popular terms like heterogeneity of data models, disparities of type-dependent storages, object-relational impedance mismatch, complexity in supporting high-order relations, and difficulty in accommodating dynamically varying data model properties.

Although issues of disparate data model integration have frequently been raised across the IT fields, for the last several decades, the focus of database research within industry and academia has been on speed, data size, stability, and integrated development environments. However, as the diversity of data objects and miscellaneous user demands grows more complex, a proper data model and its back-end storage system becomes a serious problem

of information integration in IT companies. Oracle[1] measured such expenses that by some estimates 40% amount of a typical IT budget is consumed by information integration efforts [106]. This expensive phenomenon is now widely understood as an important clue to reducing operating costs and to shortening the time-to-market (TTM). Elliotte Harold, working in the information retrieval field, also stated the challenge as[2]:

> *The problem is while much data and many applications fit very neatly into tables, even more data does not. Many other applications in fields like publishing have not even had a database. It is not that they did not need one. It is just that the databases of the day could not handle their needs.*

In a similar context, but based on multimedia aspects, we had the opportunity to conduct multimedia archiving projects on various challenging topics. Our early research goal was to improve multimedia analytical results so as to abstract low-level features from various sources. When we tried to build a summary of complex events in multimedia and hypertext sources [76, 79, 123, 71], we realized that popular information storage systems, their information object models, and their accompanying rules for association were not suited for our application. Thus, we shifted our focus to development of a new data model capable of unified information management [75, 77] that became the subject of the work presented here.

1.2 Challenging problems

Let us review two applications, both of which desperately need a new data model, from the viewpoint that the current growth rate of multimedia service applications far exceeds the progress in database models.

[1] Oracle is the trademark of Oracle Corporation.
[2] Excerpted from http://cafe.elharo.com/xml/the-state-of-native-xml-databases

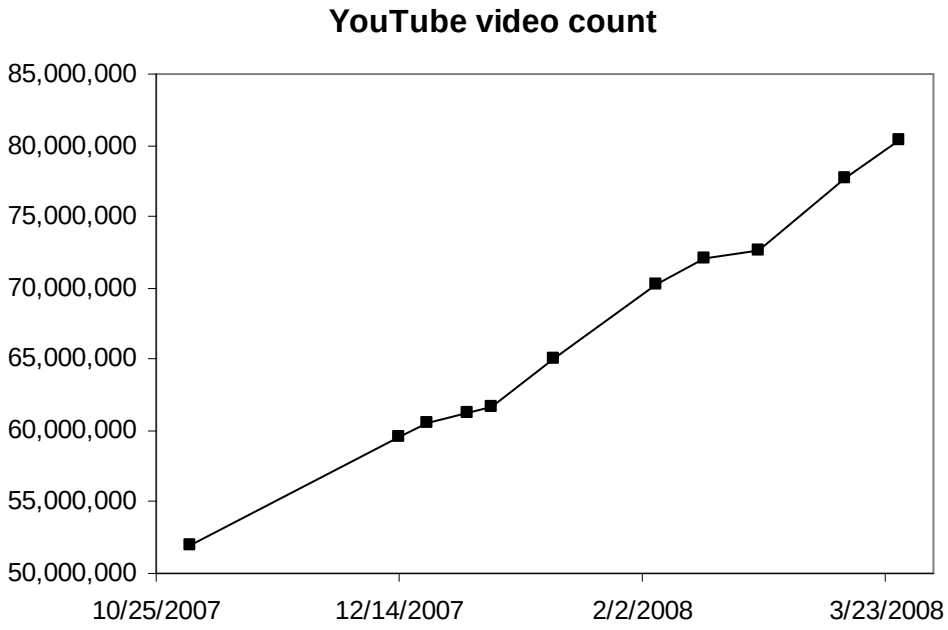


Figure 1: Total video count of YouTube increases 55.8 percent (29 million videos) in five months from November 2007 to March 2008.

1.2.1 YouTube to merge content abstractions and relations

YouTube, which is the most popular video sharing website, allows users to upload, view and share video clips. Its amazing growth can be illustrated by its 55.8 percent increase (29 million videos) from November 2007 to March 2008 in total number of videos (See Figure 1). Despite this fast growth, the structure of metadata on video objects remains primitive, confined to one of 14 predefined categories[3] and to manual tags so as to fit these videos into the structured relational database. Apparently having only two types of metadata on a video object is too limited to represent the identity and the contents of a video.

Besides, the relations with other video objects are based on an internal link to the other video ID, a linkage that simply maps one video directly to the other on the basis of their unique ID numbers. However, many users want to see exactly which portions of video

[3]YouTube video categories: Autos & Vehicles, Comedy, Education, Entertainment, Film & Animation, Howto & Style, Music, News & Politics, Nonprofits & Activism, People & Blogs, Pets & Animals, Science & Technology, Sports, Travel & Events

objects are related, and how, with which parts of the other video. Users have consistently evidenced demands for such rich semantic indexing and higher-level relation supports for multimedia content.

To analyze this problem in detail, let us look back on the source data object property. A video is a sequence of image frames with temporally synchronized multiple audio streams. Thus, all data objects with associated or superimposed information naturally embed the temporal property. When merging this source object with information on its content like scripts, cast member information, user reviews or background information, such data content objects obviously are not confined to a specific data structure. To link the source object to content descriptions, some parts of the source data should be selected by using, for instance, a spatio-temporal region selector, and they are linked to descriptions with various relations representing the description type or the semantic meaning of the role of the connection. Thus a data model for this problem should be flexible and abstract enough to employ disparate data models and to provide a generalized view on related data objects.

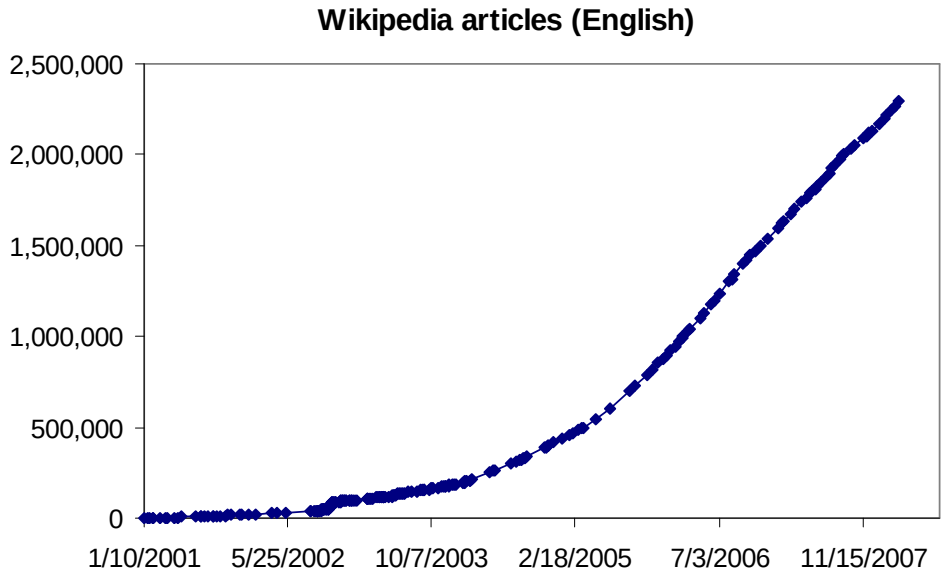


Figure 2: The growth rate of Wikipedia articles in English.

1.2.2 Wikipedia to mingle relational and hierarchical instances

A similar problem is also found in hypertext applications. As a second example, Wikipedia[4], is a free, multilingual and open content encyclopedia project. All articles are manually edited by millions of anonymous users. The growth rate of Wikipedia pages[5] has been almost exponential and now maintains a high growth rate as depicted in Figure 2. With such an amazing growth of Wikipedia, one concern is that interrelations between articles are manually made by users, and their links are based on the article title - not on the content that is the actual goal of a link. The problem occurs because of numerous ambiguous and duplicated page titles. Wikipedia describes this problem as *Disambiguation*[6], which is the process of resolving conflicts that occur in Wikipedia article titles when a single term can be associated with more than one topic, making that term likely to be the natural title for more than one article. In other words, it is guidance to forward the page link to the correct article.

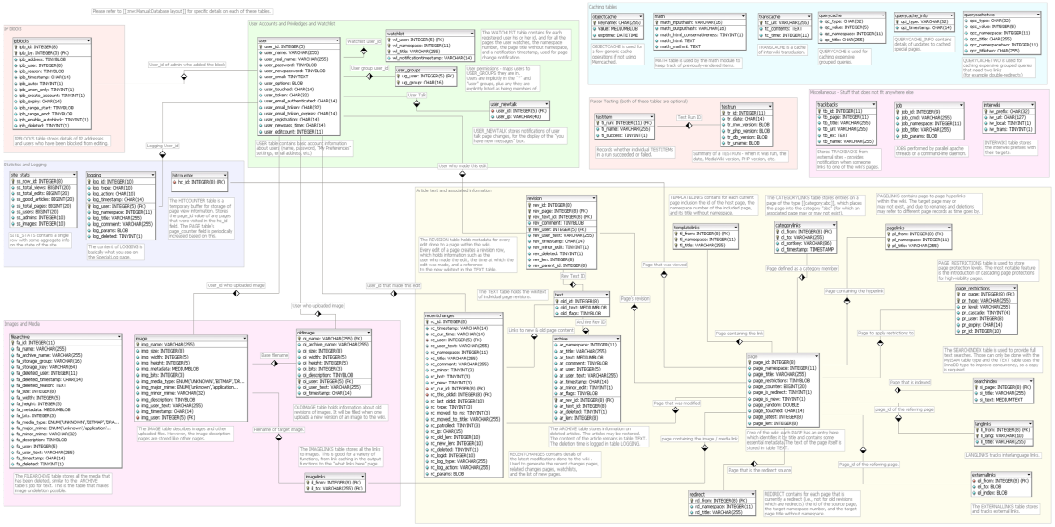


Figure 3: The MediaWiki database layout.

The inherent problem of disambiguation lies in using a page title as the identity of a

[4]http://en.wikipedia.org/wiki/Wikipedia
[5]http://en.wikipedia.org/wiki/Wikipedia:Size_of_Wikipedia
[6]http://en.wikipedia.org/wiki/Dab_page

data object. A Wikipedia article is a complex data object. If semantic meanings and distinct properties were used to identify its content, then such identity conflicts would rarely occur. However, an analysis shows why such clarity of identification is difficult with the Wikipedia database system, which runs on MediaWiki[7] software. MediaWiki is a Web-based Wiki software application written in the PHP programming language. It can use either a MySQL or PostgreSQL relational database management system. Its database layout, depicted in Figure 3, describes the complete Wiki structure with 36 relational tables.

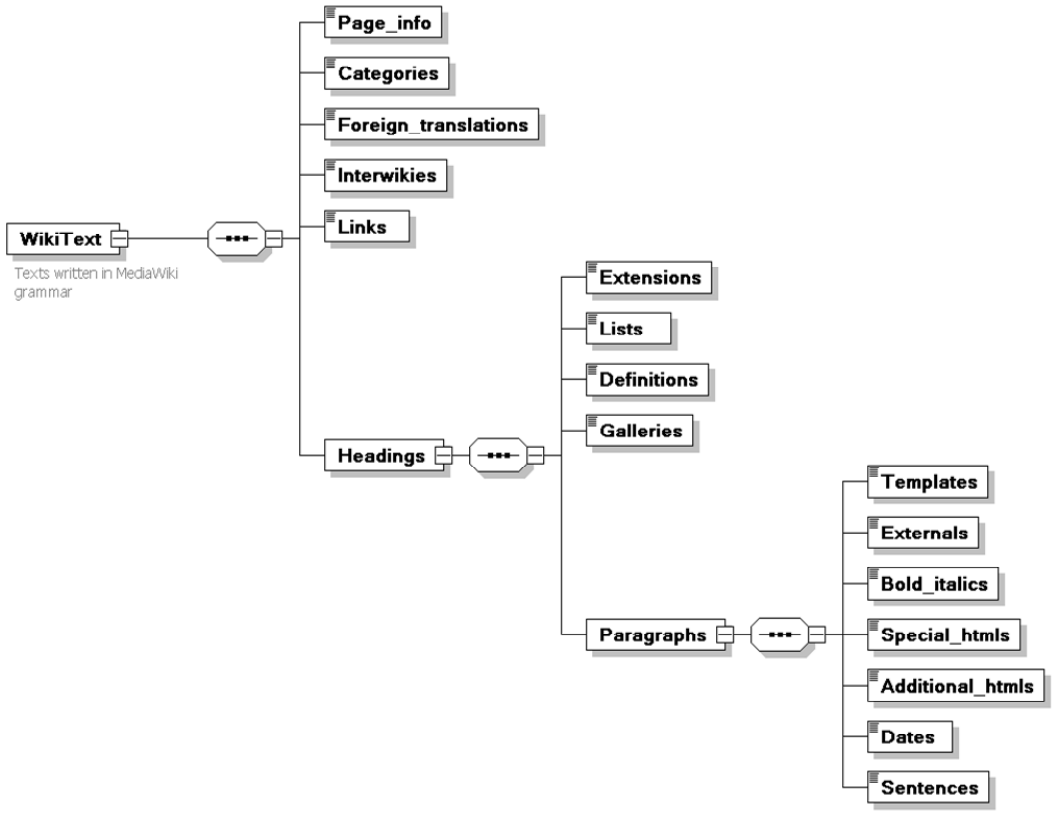


Figure 4: The WikiText schema.

In the Figure 3 database layout[8], each Wiki article is identified by its title text. Thus, if we want to change this as described above, we should analyze each Wiki article by its content and structure. A Wiki article is written in a proprietary language named WikiText. Wikitext language or wiki markup is a markup language that offers a simplified alternative

[7] http://www.mediawiki.org/wiki/MediaWiki
[8] Excerpted from http://www.mediawiki.org/wiki/Manual:Database_layout

to HTML and is used to write pages on Wiki web sites such as Wikipedia[9]. Figure 4 shows the hierarchical structure of Wikitext, which is understood by our WikiText parser. Now the goal is the integration of the hierarchical WikiText schema in Figure 4 with instances of the structured relational MediaWiki schema in Figure 3.

Modifying the structure of established archives to accommodate disparate data models requires significant effort. Combining two data models, from the above example, suffers structural disparity because unlike the relational MediaWiki database layout, the structure of a Wiki article is hierarchical, which is a generic document structure. Besides, instances of the MediaWiki database are grouped by tuples (also called ordered lists), but Wikitexts lack such grouping properties and instead are linked by hierarchical substance relations from the root instance.

This structural disparity causes problems in linking data objects across over two data models. In a relational database, a relation can only occur between two column objects, not between tables or databases; however, in hierarchical models like XML **[13]**, leaf elements or parent composite elements can both serve as objects for relation. Moreover, each data model uses a different query language like SQL or XPath. This means that an application would be overly complicated if a query had to be written in a different language to retrieve related data from each disparate database.

1.2.3 Reviews of challenging problems

The above examples illustrate the urgent need of the IT industry to manage heterogeneous data sources. As a summary, the YouTube case explains the difficulties involved in even attaching user comments and creating relations between spatio-temporally varying multi-media objects. The Wikipedia case reveals the problems in object identification, content analysis, and updating established database instances. Both cases demonstrate the necessity for a new data model capable of better service.

[9] http://en.wikipedia.org/wiki/Wikitext

If disparate data models are to be the rule, one solution that might significantly reduce the workload would be to set up a main database server to monitor all companion database systems. This server would act as a proxy server to unify communication and query methods. All queries would go through this server and then be distributed or converted to collect objects of interest. The operation of existing database servers would not need to be altered. Thus the query for a specific server would be forwarded as before. A query that searches for data objects from any databases would be parsed by the main database. This main database should have abstract and flexible ways to employ various data models of its companion databases. In the next section, we will examine the design process from the standpoint of how such a new database system might be implemented.

1.3 Data model design process

In digitized computing, we assume that an interested event is sampled and converted into discrete data objects. The resolution of the sampling rate and how to digitize them are application dependant factors left for developers. This work focuses on the data modeling approach to the process after the sampling step.

The data model design is composed of iterative steps as illustrated in Figure 5, which shows an abstract job flow in database development and management. The lower part of the flow diagram shows an iterative data model design process. The upper part is the sequence of events being archived into the database system. During the operation (the loop in the diagram), if the structure of input data is changed (like the moment t_A, t_B, t_C, t_D in the above timeline), then the schema of the databases should be updated accordingly.

Initial schema design At the first stage of a model design, a database designer would try to figure out fundamental object patterns from the source information flow. This is finding the dominant entities and the primary facts of an event. If such facts can be materialized for processing with their relations with other facts, then we call such a pattern a schema as the structure of data objects. Schema data objects are objects on data objects.

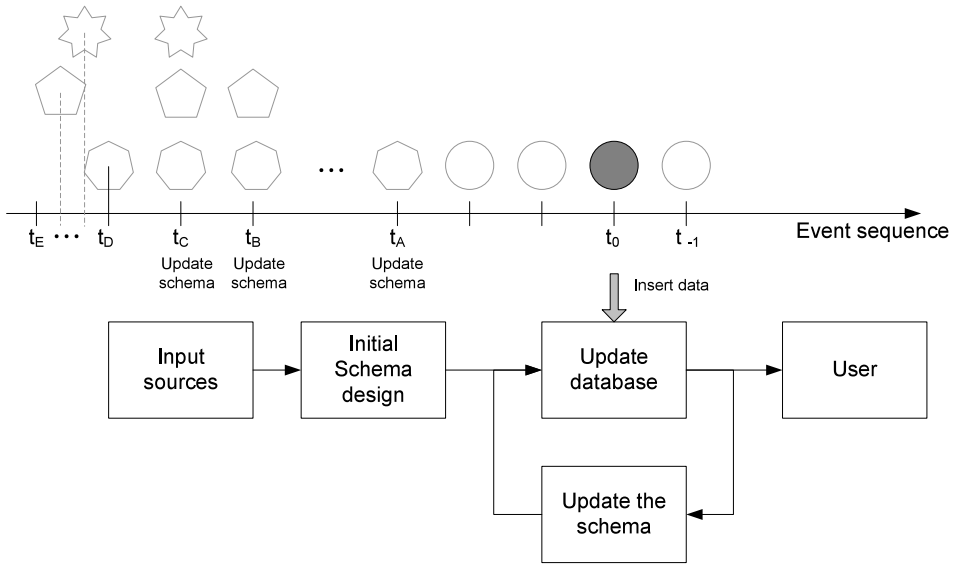


Figure 5: Data model design and maintenance process.

Hence a schema can also be represented as a data object. Such data objects are often called metadata. For instance, a relational database that supports the SQL standard [42]has the *INFORMATION SCHEMA* database that has a set of read-only views to provide information about all databases, tables, views, columns, and procedures on the database server. For XML, as an example of a semistructured model, an XML schema is the W3C recommendation [45]. It is a set of rules to which an XML document must conform to be valid and to be assured that it will be parsed by any XML parser that supports the standard. At this stage, selecting a proper data model is most important.

Update database Based on the schema structure type and the way relations are made between instances, a database designer chooses a proper database system for the data backend. Depending on the data structure, a variety of data models such as a hierarchical, network, relational, dimensional, or object data model can be applied. Once the database system is chosen, all data object schemas are interpreted to prepare the data storage. Then the system starts archiving instances of events. If a system needs to employ other data structures like the moments t_B and t_C in Figure 5, then additional data systems that fit into

new data models should be added. In the field of databases, this situation occurs frequently to accommodate the system to changes from the sources and also to meet user demands. These issues of system maintenance and application updates relate directly to the costs of operating the system.

Update the schema Because of the nature of information, data objects and their properties evolve over time. Figure 5 shows several cases of such schema changes:

CH-1 t_A represents the moment to update the existing schema. Updating schema causes modification of existing data structures. At this step, a major concern is managing data objects stored alongside new data objects in the old schema structure. This difficulty is called the schema versioning or schema morphism problem [51, 116, 118]. Some approaches just leave the newly updated schema objects as Null ; some others separate storage objects in a different version.

CH-2 t_B and t_C are moments to add disparate data models. Heterogeneous data models need each disparate storage engine. Two major problems in managing disparate database systems are (1) maintaining consistency between instances over disparate database systems and (2) updating and managing multiple queries in different languages for each application.

CH-3 Choosing the proper temporal or spatial granularity in dynamic data models is a convenient way to group instances according to their fundamental properties. This is important for a case like t_D and t_E in which the temporal correlation is difficult to apply for grouping. If such a relation is not available at the fundamental property level, then higher-level relations from the properties of each event should be checked; however, this increases the level of complexity because the database system now needs to handle disparate data types and use various methods to access and update values.

Selecting a proper data model at step CH-1 requires careful review of several design aspects: (1) costs for system development and maintenance, (2) efforts to model real- world information, and (3) limits in extending or modifying established schemas.

In addition to well-known relational models, let us review two other models. A semistructured data model permits interpretation of stored data objects without a schema definition. XML is an example of such a semistructured data model. The tremendous growth of XML in recent decades illustrates the growth of unstructured data objects that paradoxically need to be stored in a structured database system. It should be noted that if XML documents are wholly processed based on the XML schema, then their data model is structured, and in fact, text-abundant XML representation is not efficient in such cases.

Unstructured data means the model of the data object is unknown or too complex. For instance, the semantic content in one picture is considered unstructured because any objects taken in a picture could have multifarious properties with various relations with other objects. Besides, text documents written in natural language are in fact structured because they are composed and written based on the grammar in common use. However, their structural elements such as paragraphs, sentences, words, their parts, and their relations with other elements are too diverse and complex to fit into such a set-based data model. A structured model is not efficient in storing and processing a natural language, and hence, text documents are equally classified as unstructured data.

One important observation can be derived from the above descriptions. The final step of product design is a structured data model to store data objects, i.e., filtered outputs from the input sensing stream without regard to the context of their origins and intermediate changes. If data objects from the origin to the final type are stored into some generalized storage, then changes at the last structured data model can be traced exactly from the changes to the root of unstructured data objects. Such a data model should incorporate unstructured and structured data objects. Then any changes in the input stream that might cause schema morphism will not happen to a generalized data storage in which a change can be traced

to its origin. A data model for generalized data storage, therefore, will store the original sensor data input to the structured or abstracted final data objects. Its role is to serve as an archive that stores all information without loss. Various information objects exist in a real world. Consequently, the formulation and theoretical foundation of a new data model need to be developed in the sound manner that we used in our investigation in this work.

CHAPTER II

REVIEWS OF DATA MODELS

The basic terms and definitions of data model design are introduced in Section 2.1. Each data model has pros and cons that determine which application domains are suitable. Data objects and the role of each data model are compared in Section 2.2. We compared popular data models with a design case example in Section 2.3. The details of a graph data model that we developed in this work are introduced in Section. 2.4. User needs and the accompanying challenges are addressed in Section 2.5. A long-time information archive should accommodate the changes in the data schema. Section 2.6 reviews related work that uses one database system to manage multiple data models. Finally, based on the reviews in this chapter, we set up in Section 2.7 the research goals of this work.

2.1 Terms and definitions

Assuming there are series of data objects to store and query, we need concrete typing rules to model data objects and back-end storage to process data. Let us first define two aspects of a data object: (1) a discrete and (2) an identifiable object. By discrete we mean that a data object is digitized and transformed so as to be interpretable by a database. *Identifiable* indicates that each data object is uniquely accessible by an application. In this context, the series of data objects, $\overline{\mathcal{I}}$, is a set of discrete data objects, $\overline{\theta} = [\theta_1, ..., \theta_M]$ with relations between data objects, $\overline{\mathcal{R}} = [\mathcal{R}_1, \mathcal{R}_2, ..., \mathcal{R}_N]$, where $\mathcal{R}$ represents a set of relations. Thus $\mathcal{I}$ is a set of objects and their relations:

$$\mathcal{I} = [\overline{\theta}, \overline{\mathcal{R}}]. \tag{1}$$

The count of data objects, $N = |\overline{\theta}|$, where $||$ is the cardinality of $\overline{\theta}$, a finite positive integer: $0 < N \ll \infty$. The count of relations, $M = \overline{\mathcal{R}}$, is assumed a non-negative integer: $0 \leq M \ll$

∞.

The structure of θ depends on the data model. To be free from such data model dependency, abstract typing rules are applied to data objects. As the first, an *atomic* object, θ, is composed of two predicates[1]:

$$\overline{\theta} = \{\Phi_\delta(\theta), \Phi_\rho(\theta)\}, \tag{2}$$

where Φ is a propositional function that extracts a specified property of θ: Φ_δ outputs a name (symbol) of an associated data object, and Φ_ρ is a set of data object properties. This functional representation, Φ, abstracts a procedure on the part that could differ by data model. For an instance of a relational table, assuming θ is a cell object, then $\Phi_\delta(\theta)$ is a selected field name and $\Phi_\rho(\theta)$ is a set of properties like a field data type $\Phi_\nu(\theta)$, its constraints $\Phi_\varphi(\theta)$ and an actual data value $\Phi_\upsilon(\theta)$. Hence, for a relational model,, $\Phi_\rho = \{\Phi_\nu(\theta), \Phi_\varphi(\theta), \Phi_\upsilon(\theta)\}$.

In data models, θ may exist in multiple types. In relational models[30, 31, 32], *records, fields, tables, databases* and *the database of databases* are such θ objects. In an object-oriented data model [1, 98, 129], *entities, objects, instances* and *classes* in θ. In a graphical data model [7, 57, 108], *nodes, edges* and *a group of nodes* could be θ. Because there are many object class types, let us define a subpropositional function, $\Phi_\nu \in \Phi_\rho$, that identifies the object class of θ.

Each data model differs in how it implements the method to link θs. In a relational model, the relation between two θs is limited to the field-type object between two tables. Their binary relation is directional (i.e., a foreign key from the source field in one table to the target field of the other table). Such a relation can be represented in an ordered set:

$$\mathcal{R} = (\theta_s, \theta_t), \tag{3}$$

where $1 \leq s, t \leq N$, $s \neq t$, $\Phi_\nu(\theta_s) = \Phi_\nu(\theta_t) =$ "*field*". A θ is a field of a table. θ_s is the

[1] A predicate can have the role of either a property or a relation between entities.

source field and θ_t is the target field.

In a relational database, the relation between two data objects cannot be labeled because only one relation type (i.e., *foreign key*) exists, so it is meaningless to specify that in $\mathcal{R}$.. In the case of the hierarchical data model, at least two relation types are necessary: (1) parent-child relation and (2) sibling-order relation. In a tree structure, one parent element may have many child elements. A parent-child relation represents the hierarchical order like a tree-structured XML document. The order of child elements (ex., the first child, the second child, etc.) is the sibling order. In an object-oriented approach, we need more types of relations like *inheritance*, *encapsulation*, *abstraction* and *polymorphism* [1]. Hence in general, $\mathcal{R}$ can be represented for M types of relations as:

$$\mathcal{R} = (\theta_s, \theta_t, \zeta), \tag{4}$$

where $\zeta = (r_1, r_2, ..., r_K)$, r is a relation object and K is a positive integer. Now, we can derive the formal definition of a data object $\mathcal{I}$.

Definition 2.1. A data object, $\mathcal{I}$, is an ordered list satisfying the following rules:

$$\begin{aligned} \mathcal{I} &= [\overline{\theta}, \overline{\mathcal{R}}], \\ \theta_i|_{1\le i\le N} &= \{\Phi_\delta(\theta_i), \Phi_\rho(\theta_i)\}, \theta_i \in \overline{\theta}, \\ \mathcal{R}_j|_{1\le j\le M} &= (\theta_s, \theta_t, \zeta)|_{1\le s,t\le N}, \mathcal{R}_j \in \overline{\mathcal{R}}, \\ \zeta &= (r_1, r_2, ..., r_K). \end{aligned} \tag{5}$$

In Eq. 5, $\mathcal{I}$ is represented as a set of digitized objects with finite relations. Their connections are stored in the ordered list. This list is directional from source data objects θ_s to target data objects θ_t.

2.2 Reviews of data models

Each type of data model differs in how data objects and relations between them are defined. Several factors [7] influence such implementation: (1) source data properties, (2)

theoretical tools and implemented methods, and (3) hardware and software constraints. In this context, this section categorizes data models according to three aspects: structured, semistructured, and graph. These classifications are determined according to how they handle a structure of data objects that is understood as a schema of data objects. Let us first define the schema.

A schema is used to specify structure [130]. When an $\mathcal{I}$ in Definition 2.1 in or includes structured objects, such a structure is called a *schema*. For instance in a relational database, a schema defines tables, the fields of each table, and relations between a pair of fields from two tables. It also includes constraints on types of data objects and their values. A schema can be considered as a collection of data types and their relationships. Thus a schema definition can be deduced from the $\mathcal{I}$ as follows:

Definition 2.2. A schema $\mathcal{S}$ is a structure of data objects composed of a set of names, types and relations.

$$\begin{aligned} \mathcal{S} &= \{\Phi_\delta(\overline{\theta}), \Phi_\tau(\overline{\theta}), \overline{\mathcal{R}}\}, \\ \mathcal{R}_j|_{1\le j\le M} &= (\theta_s, \theta_t, \zeta_k)|_{1\le s,t\le N, 1\le k\le M}, \\ \mathcal{R}_j \in \overline{\mathcal{R}}, \quad \theta_s \in \overline{\theta}, \quad \theta_t &\in \overline{\theta}, \\ \zeta_k|_{1\le k\le M} &\in (r_1, r_2, ..., r_M). \end{aligned} \tag{6}$$

A schema $\mathcal{S}$ includes types of data objects that are necessary to create storage and to process structured data objects. A set of actual data values that conforms to the schema is called an instance of $\mathcal{S}$. A data object $\mathcal{I}$ is then a union of a schema $\mathcal{S}$ and its instance as follows.

$$\begin{aligned} \mathcal{I} &= [\overline{\breve{\theta}}, \overline{\mathcal{S}}], \\ \breve{\theta}_i|_{1\le i\le N} &= \{\Phi_\delta(\theta_i) \wedge \Phi_\rho(\theta_i) \wedge \neg\Phi_\upsilon(\theta_i)\}, \theta_i \in \overline{\theta_i}, \end{aligned} \tag{7}$$

where $\wedge$ is a binary logical connective representing a union of two data sets and $\neg$ is a logical *Not*.

The use of schema $\mathcal{S}$ classifies a data models into three types: (1) structured, (2) semistructured, and (3) unstructured.

Structured data model In a structured data model, a query should conform to the predefined schema to interpret and manipulate data objects. Many optimization techniques have been developed with constraints on structure and on data normalization. Thus database systems for structured data models focus on system performance in terms of pro-cessing speed and storage efficiency.

Semi-structured data model A semistructured data model like XML [13, 85, 108] has a preliminary embedded architecture to identify connecting elements. For structured access for computer-automated analysis, XML has an industrial standard in preparing the XML schema [45]. Besides, a XML document itself is a human-readable data object by its inherent tree architecture and its use of standardized grammar in preparing a XML document. Thus, it is possible to convey data objects to the other user without a schema definition. Thus, it is called a semistructured data model.

Unstructured data model In an unstructured data model, the schema is not defined. In such a case, a data object could be either a free form or too complex to define the schema. Multimedia content or the text content of a document is commonly referred as unstructured. For an unstructured data model, such a query cannot be precalculated. Thus, the query is prepared by value-based retrieval and navigation through the network.

Let us introduce one more data model - a graph data model that we developed in this work based on the graph theory. Because of its graph-based structure, a graph data model is more flexible than the XML tree structure. A graph data model also has a schema on its instances. Although a graph instance may not be represented in texts like XML, it is also possible to recognize data structure by walking through its connected edges. Unlike a tree-structured XML, a graph data model can represent a multiple-labeled network-structured

data model. Thus, a graph data model is considered a rich data model.

In terms of database system implementation, a structured data model provides a good separation between data representation and actual data storage. By representation, we mean the appearance that a user recognizes as its data structure. For instance, a relational model shows its data objects in a tabular form, and their connections are made through foreign keys. In a relation model, such representation is unaffected by how actual data objects are stored on the file system. This separation allows any kind of optimization techniques to be freely adopted to reduce the storage size and enhance the query speed on data files. This facilitates relational databases perform fast for structured queries.

Similar approaches can be made on the semistructured data model in case such a model is associated with a predefined schema. For instance, an XML data model stores its data in the form of a tree, and connections between XML documents are made via XInclude [99], XPointer [38] or XLink [39]. In doing so, their structure should follow the standardized definition named the XML Schema [45]. When an XML document embeds a XML schema, it can be efficiently processed using XML proprietary indexing techniques and queried in a formal way using XPath [15]. If an XML schema is not available and a user has only data instance parts (i.e., only an XML document), then a user must navigate through a tree and from a given source node through other nodes to find the target nodes. In this way, XML keeps the data object in a human-readable text form for easy interpretation in case of a graph data search. Hence, XML is a semistructured data model.

A graph data model is applied when information about data interconnectivity or topology is as important as the data itself (ex., natural language processing, multimedia analysis, chemical structure database, etc.) [7]. In these applications, data objects and relations between data objects are usually at the same level of importance. In a graph data model, relations are generally represented as functions (i.e., directed source-to-target relations) from input to output.

2.3 Data model design case study

This section designs the schema of a video-tagging example. It uses actual implementation of popular data models to get some insight into their features and limitations.

2.3.1 Example case

When we watch or hear a video, it seems natural that we would want richer experiences through more information contained in the content. However, a data structure capable of annotating multimedia is not that simple because the part of a video that appeals to a user is so diverse. Plain predicates like *authors, creation date, modified date* or *categories* without details of the content or context are too simple to satisfy such demands. If *tags* are available, they should be attached to exact locations and times for where and when they are meant to convey details on content. For instance, in content analysis a video application may use user tags or programmatic analysis results that should be confined to a region within a selected time range.

In this context, YouTube[2] recently started a video annotation service by which a user can manually specify a rectangular region for a selected time duration (See Figure 6). As of June 2008, YouTube supports various features for video tagging: (1) rectangular-shaped region selectors, (2) user tags, and (3) temporal durations. Each tag may have a link to related YouTube videos, channels, or search results. So it supports multiple storylines linked from the source video to other videos (viewers click to choose the next scene).

Let us diagnose this YouTube case. A video file $\mathcal{I}_v$ is a composite set of audio streams $\mathcal{I}_a$ and moving pictures $\mathcal{I}_p$ (i.e. frames) in which both are temporally synchronized:

$$\mathcal{I}_v = \bigcup_{1 \le t \le N} \{\mathcal{I}_{a_t}, \mathcal{I}_{p_t}, t\}, \tag{8}$$

where a time index t is an index of a digitized frame (total N) and $\bigcup$ represents the union of sets. For all time indexes t there exists $\mathcal{I}_{a_t}$ and $\mathcal{I}_{p_t}$, each of which which is a subset of

[2] YouTube: http://www.youtube.com

$\mathcal{I}_a$ and $\mathcal{I}_p$. This can be represented in first-order logic (FOL) [70],

$$\forall t \exists \mathcal{I}_{a_t} \exists \mathcal{I}_{p_t} ((\mathcal{I}_{a_t} \in \mathcal{I}_a) \wedge (\mathcal{I}_{p_t} \in \mathcal{I}_p)). \quad (9)$$

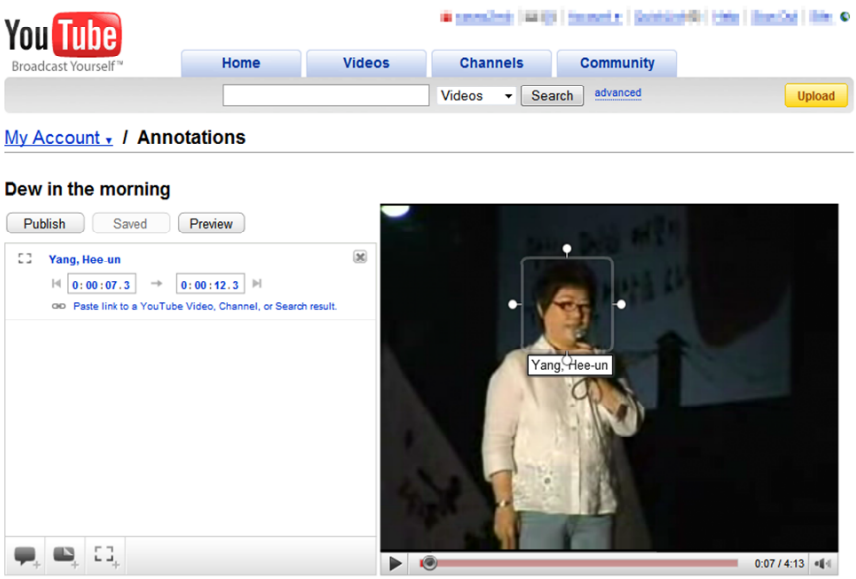


Figure 6: YouTube video annotation interface.

Since $\mathcal{I}_a$ and $\mathcal{I}_p$ are digitized in different temporal granularity (ex. $\mathcal{I}_a$ in 128 Kbits/second and $\mathcal{I}_p$ in 30 frames/second), let us simplify the problem by assuming that $\mathcal{I}_{a_t}$ is an audio segment of a matching video frame $\mathcal{I}_{p_t}$.

To link related data objects to some parts of $\mathcal{I}_v$, we need to select a region $\mathcal{G}$ within $\mathcal{I}_v$. Since $\mathcal{I}_v$ is composed of two data object types (image frames and audio streams), two region types are necessary for each type. For $\mathcal{I}_a$, because of its one-dimensional property, its region $\mathcal{G}_a$ is a temporal range specification from start time to end time. For $\mathcal{I}_p$, its selection $\mathcal{G}_p$ is a spatial area specification based on its two-dimensional property. In fact, $\mathcal{G}_p$ can be in various geometry types, depending on its spatial object property. This work uses standard spatial geometry types named *Geometry* as defined by OpenGIS for SQL [105]. *Geometry* includes *Point*, *Curve*, or *Polygon* like shapes for region specification.

Multiple $\mathcal{G}$ objects can be attached to one $\mathcal{I}_{p_t}$. each $\mathcal{G}$ has a temporal duration of up to the time when the target object deviates from the region $\mathcal{G}$. This duration may exist

over multiple $\mathcal{I}_{p_t}$ Consequently, the mapping between $\mathcal{I}_p$ and $\mathcal{G}$ objects is many-to-many. Assuming the set of frames as $\mathcal{I}_P = \bigcup_{1\le t\le N} \mathcal{I}_{p_t}$ and the set of geometry object as $\mathcal{I}_g = \bigcup_{1\le i\le M} \xi_i$, the many-to-many (N-to-M) relations $\mathcal{R}_{ig}$ can be represented as a set of two objects:

$$\mathcal{R}_{ig} = \bigcup_{1\le i\le N, 1\le j\le M} (\mathcal{I}_{p_i}, \xi_j). \tag{10}$$

The next step is linking each geometry object ξ to what that marked area is meant to be. Such related data for each ξ could be other ξ objects at different times, or they could be a disparate object like an external reference to a video, a hyperlink, a spatial location (latitude and longitude), other types of binary objects such as thumbnails or PDF files, or simple text memos. In linking such heterogeneous media, the necessary relation object would be a set of:

$$\mathcal{R}_{go} = \bigcup_{1\le i\le M, 1\le j\le K} (\xi_i, \Phi_\tau(O_j)), \tag{11}$$

where Φ_τ is a functional abstraction of a selected object property.

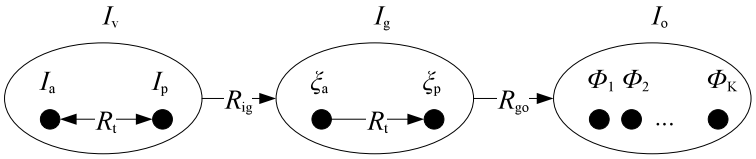


Figure 7: Multimedia annotation schema example.

This connection can be depicted as in Figure 7 in graph form with three categories: $\mathcal{I}_v$ for video objects, $\mathcal{I}_g$ for geometry objects and $\mathcal{I}_o$ in Eq. 12 for tags on objects. This graph exemplifies the schema of the above YouTube video annotation.

$$\mathcal{I}_o = \bigcup_{1\le j\le K} (\phi_1, \phi_2, ..., \phi_j, ..., \phi_K). \tag{12}$$

In $\mathcal{I}_v$, picture frames $\mathcal{I}_p$ and audio streams $\mathcal{I}_a$ are synchronized by the temporal relation $\mathcal{R}_t$. Let us name this schema $\mathcal{S}_y$. The geometry object $\mathcal{I}_g$ is a region selector in audio streams ξ_a or picture frames ξ_p. If an attached tag $\mathcal{I}_o$ merely needs a temporal selection (like a video subtitle), ξ_a could exist stand-alone and vice versa for ξ_p. The relation $\mathcal{R}_{ig}$

is the connection between $\mathcal{I}_v$ and $\mathcal{I}_g$. $\mathcal{R}_{ig}$ is a set of two indexes from two categories that makes their connection distinct.

$\mathcal{S}_y$ shows a nested-set design case of organizing objects in an object-oriented manner. For a video object category, the index key of $\mathcal{I}_v$ is a distinct composite set of $\mathcal{I}_a$ and $\mathcal{I}_p$ keys. Similar indexes could exist for $\mathcal{I}_g$ and $\mathcal{I}_o$. The way to index objects and define the relation now differs by data models. In the following sections, popular data models are used to implement the YouTube annotation schema $\mathcal{S}_y$ for comparative purposes.

2.3.2 RDBMS schema design

A relational model is based on first-order predicate logic as shown in Figure 8. A relational database is an incarnation of first-order relations in which an object formulates a rectilinear table. The focus in such relational databases is on the size and speed of the data operation instead of emphasizing the degree of freedom for the data model design. Relational databases are adequate for data that exhibit a great deal of regularity with a relatively low degree of complexity.

$$e_1 \xrightarrow{r_1} e_2 \xrightarrow{r_2} e_3$$

Figure 8: The linked list of first-order relations.

If the relational model is applied to schema $\mathcal{S}_y$ for multimedia annotation, a schema, which is a set of database and table definitions, should be predefined to fit into the structured database. Specifically, the relation $\mathcal{R}_{ig}$ needs to include two column keys to store rows of frame IDs (time index t) of $\mathcal{I}_v$ and indexes of geometry objects ξ in $\mathcal{I}_g$. To use (ξ, t) as a foreign key, a unique constraint should be applied to $\mathcal{R}_{ig}$. The next relation $\mathcal{R}_{go}$ between $\mathcal{I}_g$ and $\mathcal{I}_o$ is more complex because the types of target objects $\mathcal{I}_o$ and their entities $\Phi_\tau(\mathcal{I}_o)$ are not confined to a specific data type; moreover, the count of each entity could be different. For instance, if $\mathcal{I}_g$ is a set of hypertext objects, two elements (ϕ_1 is a hyperlink and ϕ_2 is a data type indicating that the type of ϕ_1 is a hyperlink) would be sufficient. For spatial

locations, three elements (ϕ_1 for latitude, ϕ_2 for longitude and ϕ_3 for a data type object) are necessary. Therefore additional data schemas should be added whenever a new data type is employed for multimedia content abstraction.

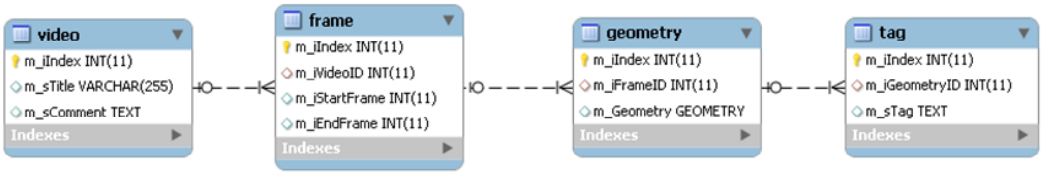


Figure 9: Video tag relational schema.

To apply the relational model on the second relation $\mathcal{R}_{go}$, one table for each data object is necessary plus an additional table to specify relations between them. Assuming the total count of distinct $\mathcal{I}_o$ as M, then the minimum number of tables $N_{t_{min}}$ would be $N_{t_{min}} = 2M+1$. Hence, assuming a worst case in which no objects share a common data type, $N_{t_{min}}$ increases at each step in a polynomial order of the depth of linked relations (N_{path}) as shown below:

$$N_{t_{min}} \propto \prod_{1 \le i \le N_{path}} N_{ob_i}. \tag{13}$$

where $N_{ob_i}, 1 \le i \le N_{path}$ is the count of distinct object types.

Thus the query efficiency will drop significantly as more tables need to join in. Accordingly data management becomes more difficult in terms of system maintenance and time costs.

2.3.3 XML schema design

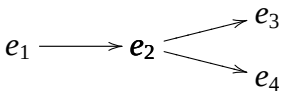


Figure 10: Tree-structured object relations.

Hierarchical data models are mostly represented using XML documents. Because one XML document has only one root node, such collections of different XML documents form

a forest of documents. Queries directed into such a forest need separate query commands for each document. Shared data objects within a document can be made through ID and IDREF, but when searching through documents that are related but have different XML schema (i.e., different hierarchical structures), additional XML extensions like XInclude, XPointer or XLink are necessary to define the references between them.

For $\mathcal{S}_y$, flexible data models like XML [13] could work better than RDBMS. Figure 11 shows an example of a XML schema design in which "shot" means those segmented video frames with start and end frame specifications. Geometry and tag objects are attached as the child elements of "shot" objects in sequence. Although XML can represent the hierarchical order between items, its schema should be changed accordingly whenever new data types are introduced for richer content abstraction. Such changes in the schema can cause interpretation problem for applications. Thus, all applications that rely on the old schema should be reprogrammed accordingly.

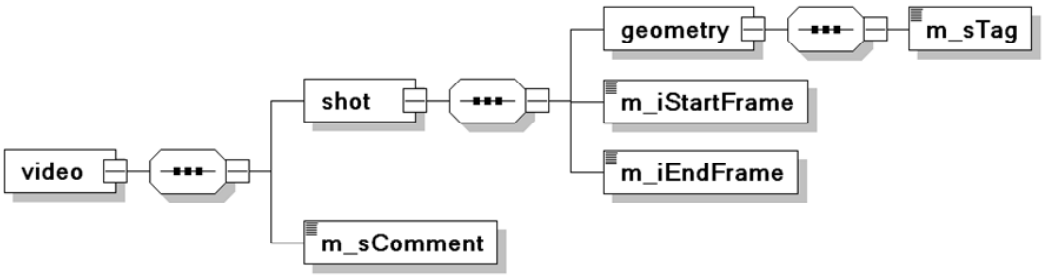


Figure 11: Video tag XML schema.

Besides, because of the tree structure of a XML document, child data instances cannot be shared. For instance, one data object in a movie could appear many times. However, to maintain its tree structure (one parent object has many child objects), such data objects need to be inserted as often as they appear in the movie. This data redundancy results in the loss of space and efficiency. Such elements also may have ID properties and be shared by using IDREF[3] tags. These three facts - the complexity in linking shared objects, the restriction of

[3]Certain kinds of XML validation need access to the document in full. For example, a DTD IDREF attribute requires that there is an element in the document that uses the given string as a DTD ID attribute.

relations to tree shapes, and finally, the limited constraints on relations between elements - negate any claim that the complexity of the retrieval problem is reduced in XML.

2.3.4 Relations in data models

Chen's entity-relation (ER) model [27] defined an event as a connector between one to many relations. The role of an event within such a definition is a flat directed connection between fixed index keys. A relation is not treated as a physical information object but as metadata on table relations. This makes connections between tables flat and permits only one-to-one connections. Thus, a relational model has to physically create a connection $N \times M$ times to support N-to-M relations. The XML model has a similar problem in which $N \times 1$ relation is not supported, and only $1 \times M$ is allowed. As shown in Figure 12, multiple attributes for one element are banned. Hence, in XML, only nested relations can simulate multiple attributes. But this increases tree depth and significantly affects query speed.

```
<family>
  <name husbandof="B" fatherof="C1" fatherof="C2">A</name>
  <name wifeof="A" motherof="C1" motherof="C2">B</name>
  <name daughterof="A" daughterof="B">C1</name>
  <name daughterof="A" daughterof="B">C2</name>
</family>
```

Figure 12: Family pedagogy represented in N-to-1 relations is not allowed in the XML schema.

2.4 Graph data model

In addition to the descriptions given above, existing data models need a priori knowledge of data schemas to search across databases. For instance,

- RDBMS: Which column of which table is it under, and with which column of the other table (of a different database) is it related?
- 2.5 XML: In which document and at where is the object of interest located? To

which object is it referenced by other IDREF objects? Is it connected to other XML elements via XInclude?

In addition to the problems mentioned previously, maintenance of databases in mixed data models requires a database expert who fully understands the DNA of the target application and can readily modify or compose queries. However, creating a mixed view of both relational database tables (a list of list models) and XML documents (hierarchical models) is not that feasible because of the conflicts in their structure and their different query methods. Designing a proper data model for multiple data structures that propagate their relations over heterogeneous data models poses huge challenges. A more generalized and flexible data model that can seam-lessly interact with these data model methods would help such design significantly.

Graph data models are well-known for their flexibility in modeling complex data models. In digitized computing, data objects and their relations with other objects can be naturally represented in a graph. Also existing data models can be readily represented in a graph data model through a set of rules on assigning nodes and edges; the exception lies with the indexing and querying methods, which are specialized for each data model. Let us first formulate a graph and its data model architecture.

Definition 2.3. A graph is a collection of vertices connected by links that are also called edges. A *directed graph* or *digraph* $E = (E^0, E^1, r, s)$ has directed links that consist of two countable sets E^0, E^1 and functions $r, s : E^1 \to E^0$. The elements of E^0 are called *vertices* (or *nodes*) and the elements of E^1 are called *edges* (or *relations*). For each edge, e, $s(e)$ is the *source* of e and $r(e)$ the *range* of e; if $s(e) = v$ and $r(e) = w$, then we also say that v *emits* e and that w *receives* e, or that e is an edge from v to w[11].

A digraph is a directed graph with no directed cycles; that is, for any vertex v, there is no nonempty directed path[4] that starts and ends on v. A *walk* in a graph is an alternating

[4]A path that includes at least one vertex

sequence of vertices and edges, beginning (*origin*) and ending (*terminus*) with a vertex in which each vertex is incident to the two edges that precede and follow it in the sequence; the vertices that precede and follow an edge are the end vertices of that edge. The length of a walk is the number of edges that it uses. A walk from a query means searching by the rule of walks over related information from a given starting vertex.

Directed graphs have been used in many applications. A city street map and an abstract representation of computer programs and network flows are applications that can be readily represented by directed graphs. Directed graphs are also used in the study of sequential machines and system analysis in control theory. In this work, we are more focused on their expressive power in modeling the directional flow of information with constraints, node definitions, and functional association rules.

A graph data models uses a set of fundamentals to define a way of creating graphs. These fundamentals are: (1) a collection of data structure types (node and edge labels and types), (2) a collection of operators (type coercion and path expression), and (3) a collection of general integrity (graph structuring) rules that implicitly or explicitly define the set of consistent graph states or changes of state or both [32]. In graph data models, data structures for the schema and instances are modeled as graphs or generalizations of them. Data manipulation is expressed by graph-oriented operations and type constructors [7].

Graph data models are generally considered more natural in representing world facts. Typically, graph data models are applied in areas in which information about data interconnectivity or topology is more important, or as important, as the data itself. In these applications, data objects and relations among them are typically considered at the same level [7]. These information domains exceed in complexity even the multimedia annotation cases discussed earlier and typically take in categories such as natural language processing, genetics, newspapers, libraries, telecommunications, and sales. Within various graph data models developed for several decades [7]. we compare features of selected graph models and discuss further improvements contributed by our model.

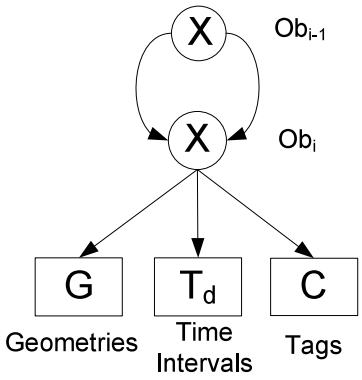


Figure 13: Logical data model schema.

Kuper and Vardi [86] proposed Logical Data Model (LDM) that describes mechanisms to restructure data and logical and algebraic query languages to process data. In LDM, a schema is an arbitrarily directed graph in which each node is one of three types: *basic*, *composition*, and *collection*. Its instances consist of two-column tables, each of which associates entities of a particular type (*primitive*, *tuple* and *set*). LDM provides logic (similar to relational tuple calculus) to specify views and integrity constraints on LDM schemas. Integrity constraints are LDM formulas. These formulas enforce requirements that instance objects satisfy certain conditions (satisfaction of LDM formulas). That is, given a database and a sentence in the logic, one can effectively test whether the sentence is true in the database instances. Figure 13 shows the LDM schema design of the YouTube video annotation sample contained in Figure 6. This schema uses three basic type nodes for representing data values (*Geometry*, *Time interval*, *Tag*). The instance is a collection of tables, one for each node in the schema. In LDM, edges are not labeled.

GROOVY [93] is an object-oriented data model that is formalized by using hypergraphs. A hypergraph is a generalization of a graph in which edges can connect any number of vertices. Formally, a hypergraph H is a pair of (X, E) where X is a set of elements called nodes or vertices, and E is a set of nonempty subsets of X called hyperedges or links. Therefore, E is a subset of $P(X)$, where $P(X)$ is a power set of X. While graph edges are pairs of nodes, hyperedges are arbitrary sets of nodes and therefore, can contain an arbitrary

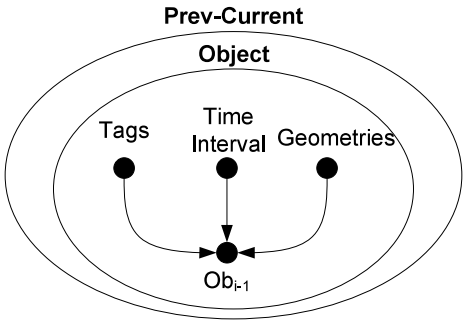


Figure 14: GROOVY schema.

number of nodes.

GROOVY is featured in the use of hypergraphs as a formalism for modeling complex objects, sub-object sharing, integrity constraints, and structural inheritance. Although GROOVY does not use labeled edges, the task of representing relations (and their names) can be undertaken by encapsulating edges, which represent the same relation (same label edges), within one hypernode (or hyperedge) labeled with the relation-name. A GROOVY provides support for nested structures. A novel feature of GROOVY is the use of hypergraphs to define value functional dependencies. The hypernode model is characterized by using nested graphs at the schema and instance levels.

In the GROOVY schema at Figure 14, an object is modeled as a hypergraph that relates the attributes Geometry, Time interval, and Tag. The value functional dependency (VDF) is logically represented by the directed hyperedge. This VDF asserts that Geometry, Time interval, and Tag uniquely determine the set of Previous Objects. In GROOVY, edges are not labeled. The GROOVY model shows how structural inheritance is supported naturally by nested graph structures. GROOVY introduces the notion of object-class schemas over which objects are defined. An object schema defines valid objects (value-schemas), value functional dependencies, and valid shared values among objects (sub-object schemas). These restrictions are formalized using a hypergraph representation. Indeed, there is a one-to-one correspondence between each object schema and a hypergraph in which objects are vertices, value functional dependencies are directed hyperedges, and sub-object

schemas are undirected hyperedges.

Note that in the GROOVY model, the three object type classifications (*basic*, *composition*, *collection*) used in LDM do not appear. This is because in GROOVY structural relationships are represented by an enclosed hypernode in which other objects can freely reside. As a result, the distinction between composition and collection types cannot be represented with this hypernode notation.

GROOVY uses directed hyperedges to represent VFDs, which are used in the value schema level to establish semantic integrity constraints. A VFD asserts that the object value restricted to a set of attributes uniquely determines the object value restricted to a further attribute.

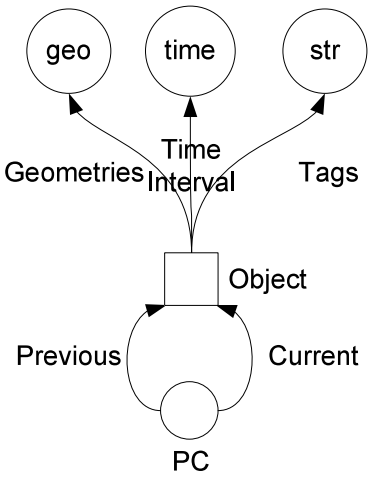


Figure 15: GDM schema.

GDM [67] is a graph-based db-model based on GOOD, which adds explicit complex values, inheritance, and n-ary symmetric relationships. Schema and instances in GDM are described by labeled graphs, called instance graphs and schema graphs, respectively. In the GDM schema in Figure 15, each entity object has assigned attributes: Geometry, Time interval, and Tags. Basic value nodes are represented as round with a labeled data type. A composition value node PC establishes the Previous-Current relationship. Redundancy is introduced by the node PC. In GDM, edges and nodes are both labeled.

An extended graph data model feature comparison is provided at [7]. The E-model as a comparison can provide most of their features by its inherited general node and edge definition supports. Specifically, the E-model system is implemented with rich sets of APIs for database system implementation. Most graph data models introduced at [7] are limited to the theoretical proposal based on our contacts with the authors. We could verify the implementation of GOOD [60] but did not see proof of implementation for other graph data models. The E-model can be generalized applied to their implementation by its rich features for graph data model realization as a database system.

2.5 User needs and challenges

Users in many fields - genetics, libraries, newspapers, sales, telecommunications, etc. – continue to emphasize the necessity for a new data model In response, experts in academia and industry have proposed many data models. Some high performance models continue to enter the market. Regardless of the success of any specific model in the highly competitive database business, it is widely acknowledged that no solution can satisfy all customers.

The need for a new data model is conspicuous in information retrieval (IR). Pat Case, a U.S. Library of Congress librarian, has said [6] that most previous integration attempts have merged databases and IR engines without any fundamental changes to either . No single unified data model exists to manage both structured and unstructured data for processing both precise and ranked query results. She summarized the requests from the IR community as follows:

- A good search system must allow users to refine their search results by explicitly limiting or expanding the number of answers or by using taxonomies and ontologies.
- The ability to parameterize the scoring method used to rank query answers. Proprietary scoring algorithms on behalf of end-users and a way to rank query results are necessary. Case argued for relevance that is based on user-specified criteria, not on some word frequency method. A system should permit exact and unscored searches.

- Ordered and unordered word distance (ex. similarity, synonym, antonym, hypernym or hyponym like synset5 category) operators [24, 47].

- A standard end-user syntax or a query language that combines structured and unstructured searches and that can be used reliably across search systems.

- Ranked (i.e., ordered) result lists instead of results sets, which are especially needed and desired in customer support and health-care management. The IR paradigm of ranked retrieval is based on probabilistic models of relevance over structured data. The essence of these requests is a call for a unified ranking methodology for all kinds of combined information.

As shown in Section. 2.3, we have designed the actual schema using the popular data models. In terms of the foregoing context, let us summarize the features and limitations of the popular data models.

The relational model, which is the mathematical foundation of RDBMS, was first proposed by E. F. Codd [30, 31]. The fundamental assumption of the relational model is that all data records are represented in first-order logic (FOL) in which a flat relation is a subset of the Cartesian product of n domains. However, in practical design, several problems have been reported in [130] and within those, the limitations discussed below are notable in complex data modeling:

ER-1 The ER permits only first-order relations. This uniformity of relations is a too-strict requirement for certain types of data such as CAD (Computer-Aided Design), images, or audio data.

ER-2 A class inheritance or a subtype of a more general type cannot be modeled naturally in a relational model. For instance, an object-oriented *IsA* relation, which needs structural, semantic and operational specialization, is difficult to represent with relational models.

ER-3 Aggregation (average, minimum, maximum and counting) functions are defined only based on atomic (one reference field) relations and are not uniquely definable for views (composite fields).

Similar problems also exist in other popular data models. The hierarchical model [25] structures data as a tree of records in which each record has one parent record and many children records. The network model [10], however, allows each record to have multiple parent and child records, forming a lattice structure. The network model is also called a navigational database because its method of accessing data is that of a reader moving along paths to search data. This means the connection between data keeps the flow of information in the temporal order of occurrence. The hierarchical and network models are feasible in handling ER-1 and ER-2, but not ER-3. For ER-1, n-*ary* relations in both models can be represented by nested multiple binary relations. ER-2 can be aptly handled in both tree and network models by using their information flow to represent hierarchical inheritance.

XML was born as a protocol to exchange information via networks but now is being widely accepted as an information storage model for semistructured data. Along with such trends, native XML storages are emerging into the market because of their power of representation and data model design flexibility.

Normalization in a hierarchical model (ex. XML), is a little tricky [113] and not yet standardized. The difficulty in XML lies in the fact that index keys are put under the schema definition for a XPath selector to locate its reference. Also, the way to create a key index and its elements depends on the designer and is not under the control of algorithms. Both models support a flat first-order relation that has no meaning other than as an indication of a data connection. Hence, both models are proper applications for putting sets of data in one group and connecting them to another group. This can be easily understood as extending connections between instances such as a list of lists or a forest of trees.

2.6 Related works

To accommodate the above requests for advances in data models, hybrid relational-XML databases [107, 53, 101, 17, 102] are gaining popularity as a way to support both models from one database system. Alan Halverson, et al., [64] classified the mixed SQL and XQuery system into four categories:

- Shredding (XOR) architecture: XOR represents the XML-Over-Relational approach in which the XML documents are shredded into atomic values that are then stored in relational tables. XQuery statements are translated into SQL queries to be evaluated by the existing query processor. LegoDB [20] and XPeranto [121] offer different shredding and XPath querying capabilities based on this approach. However, none has yet managed to produce a fully compliant XQuery implementation.

- Co-processor architecture: This stores XML as unparsed text in the text columns of relational tables. The XML instances are queried using an XQuery processor that is external to the database system and invoked much like a user-defined function. This solution is attractive for its relative simplicity and modularity, and most commercial database systems [41, 44] support this type of XML manipulation. However, the entire XML document is usually brought into memory before processing, severely limiting the size of the data as well as optimization possibilities.

- Side-by-side architecture: This architecture has a tighter coupling between the query processors. Query fragments can be translated from one language to another and exchanged using internal data structures that may not adhere to the language semantics. This structure introduces many complexities that require that various system components have compatible definitions on both sides of the system.

- ROX architecture: ROX is the opposite of the XOR architecture. This architecture requires a complete XQuery engine that is adapted to run SQL queries, seemingly a

much more demanding path than the opposite route. [64] shows one implementation in which the claim is made that building SQL on top of an XQuery engine poses a significantly lesser challenge than the opposite approach.

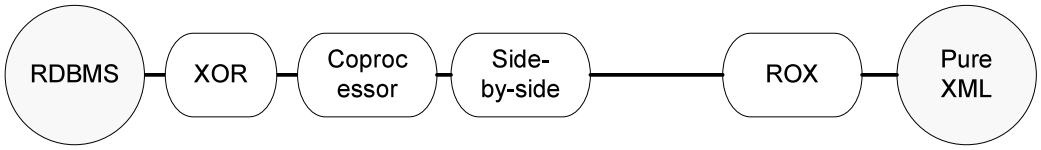


Figure 16: Hybrid data models for RDBMS and XML.

Figure 16 positioned hybrid data models by their relativity to RDBMS and to XML. These hybrid models are tuned for two data model interoperation. Technically transformations between queries are focused on the language translation between XQuery and SQL. Thus, adapting these hybrid models for newly introduced data models will not be feasible because they are based on either one or the other, or both, of the XML and relational database systems. They introduce neither new objects nor new relation techniques. In light of these limitations, adding to either data model the capability to handle text or multimedia objects, which both data models have difficulty supporting, would require extensive work.

2.7 Research goals

Figure 17 illustrates the current database status for XML and RDBMS operations that cover the top-left side of the necessary application domains. The work reported in this paper aims to develop a more generalized data model, called the E-model, for broader domain applications, including text and multimedia.

This work questions the bottom-level object and whether such bases of data units can be further improved for complex data models. By bottom-level object, we mean a primary indexing unit to access a raw data value just as for relational databases; a primary index is a way to access a record and also is a data structure to enhance access speed. The other fact is that the type of index should be independent of the data type. However typical

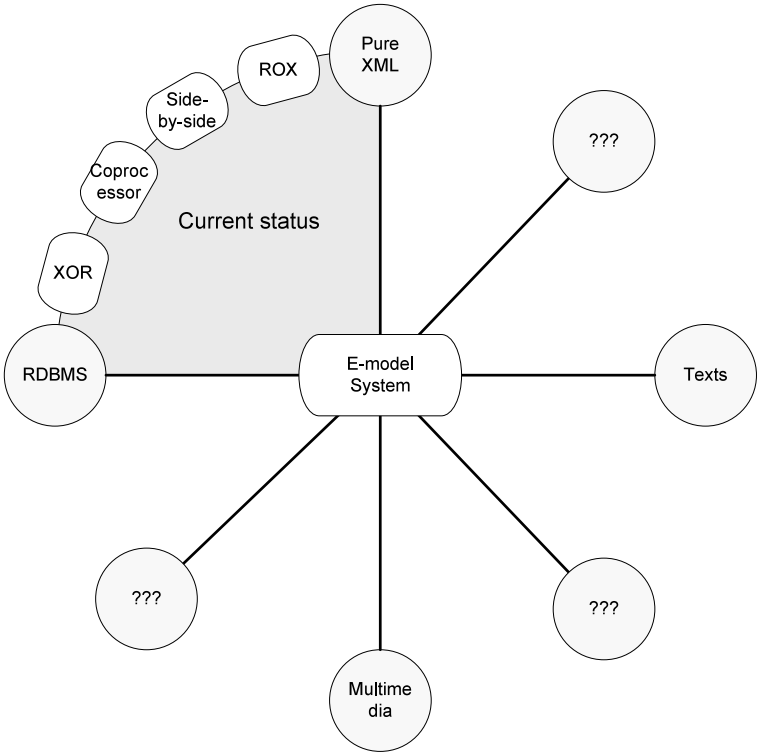


Figure 17: The role of E-model as the centralized archive for disparate data models.

indexing methods depend on the data length. Blob fields are not indexed for this reason. For instance, a raw level data index like B+ trees [12] has a presumed limit on its maximum data byte. Also, the raw data index is restricted to one table, i.e., for one fixed data model. To meet our demands, we require an indexing mechanism capable of working on multiple levels from the most basic objects up to such levels that even include schema definitions and databases. More fundamentally, we need new definitions for data models, objects, and instances. In this context, we set up our research goals (RG) as follows:

RG-1 In handling heterogenous objects, the database system should be object data type-independent.

RG-2 For networking, the unit of information transformation should be object-type and data model free.

RG-3 Queries can be made by any combination of relations on-the-fly and multiple relations can be modeled.

RG-5 Implementation of the system should be on top of existing databases. If possible, improve existing systems in addition to their well-tuned performance in handling structured information and simultaneously provide flexibility in their information storage and query functions.

Table 1 summarizes our research goals for the E-model in terms of six aspects, which are also compared with two popular data models that are implemented at the end of this work. In the table, operation units indicate the objects and protocol used to communicate with the system. Query means a language to program such communications. The goal of each database model is described in the application row.

In designing a new data model to meet the above goals, we followed as a guideline the suggestions of E. F. Codd [32]. Codd said that such an undertaking should contain (1) a collection of data structure types, (2) a collection of operators, and (3) a collection of general integrity rules that either implicitly or explicitly define a set of consistent database states or changes of state or both.

This paper accordingly illustrates a new proposed data model that is named an event-based functional data model, in short an E-model. We first define a fundamental information object named an e-node. Then we explain how such generic objects can be formulated in a way users can store and query information. Within the E-model, grouped events constitute the relational acyclic directed graph (RDAG). Directed connections conform to the natural temporal flow of events. Our work is currently moving beyond the data model toward actual implementation. Next steps and future research topics are discussed in the conclusion.

Table 1: Data model comparison chart

	RDBMS	XML	E-model
Application	Structured data	Semistructured data Documents Interoperable transmission	Heterogeneous categories Unpredictable data models Progressive archiving
Query	Structured query (SQL)	Tree-structured query (XPath)	Query by value, name, and time. Graph walk Structured query on e-categories
Join	Primary key, foreign key Flat relation No mutual relation	XInclude	Join by category Join by value and/or name Join by temporal relation
Normalization	Designer choice (redundancy vs. performance)	Possible	Fully normalized
Data order	No order but by value	Parent-child and sibling order	Directed relation, temporal order, ranking by reference count
Operation unit	Database (ODBC[5] [122])	Document (XPath, XQuery)	e-nodes
Group	Database, table	XML collection	e-category
Data	Row	XML document	e-node
Design	Fixed at the application design time	Can be changed in real time for slow Web applications	Dynamic categories in real time Dynamic views for caching and interoperation

CHAPTER III

E-MODEL

This chapter introduces the mathematical formulation of an event and its data model. An event has been adopted in various fields as reviewed in Section 3.1 To clarify and distinguish our concept among such diverse meanings, we define an event as a materialized data object named as a pure event that is the generic identity of an occurrence independent of domain factors. Our concept of a pure event coincides with that of thinkers in the philosophy of mind and metaphysics, as will be introduced in Section 3.2. We expand our concept of an event as an information object. Its entities and properties are formulated in Section 3.3.

Based on a fundamental pure event object, a new data model named the E-model is developed in Section 3.4. In this implementation, a pure event becomes an e-node composed of a set of both the name and value of a data object. The E-model distinctively uses the same object type for name and value, which is c-data and introduced in Section 3.5. C-data objects make the E-model homogenous and interchangeable. C-data objects also enable the relation between heterogeneous data models to propagate through from both symbol objects and value objects.

3.1 Events

In looking for a new data indexing mechanism, an event became the focus of our work. This focus arose from the observation that in the modern era, a moment of interest is digitized and processed by a computer. An event, ordinarily representing something that happens, refers to a significant occurrence or happening or to a social gathering or activity[1].

[1]The American Heritage Dictionary of the English Language: Fourth Edition. 2000

An event also refers to various meanings in the areas of science. Cited from the Wiki page on Event[2], an event occurs at a point in time that can be distinguished because the state of the world changes. Something was different before and after an event. An event in special relativity (and general relativity) is a point in the space-time continuum, i.e., it has a position in space and time. In experimental particle physics, an event refers to a set of elementary particle interactions recorded in a brief span of time. In probability, a possible outcome of an experiment is called an elementary event, and a set of those (a subset of all) is called simply an event. In philosophy, one might want to distinguish facts from events and also between physical and mental events. In information processing, an event is a change in the properties received by an observer after being transmitted from an object. In computer programming, an event is a software message that indicates something has happened. A number of protocols, such as MIDI[3], are also event-based. In a database, the entity-relationship model [27] mentioned an event as a connector of one-to-many relations in an abstract view.

In terms of a database, an event would be an index of a data objects set. An event for this purpose should be materialized with a specific data type and with rules of association for object definitions, relations, and attributes with constraints. A generic event concept such as high-level social gatherings or group meetings cannot fit well into databases because too many heterogeneous data models are associated with them. Thus, when any event is used as a generalized identity for any kind of happening, the associated data objects and their characteristics should not be bound to a specific data model. Although database systems support and provide systematic ways to store and retrieve structured data objects, the gap between databases and events is still wide.

We believe a storage model for such complex events should be designed in a way that will allow this model to be more general and sufficiently abstract so as to be free from

[2]http://en.wikipedia.org/w/index.php?title=Event&oldid=24109171
[3]MIDI (Musical Instrument Digital Interface): http://www.midi.org/

an embodied, enacted, or situated philosophy. Independence from any particular choice of representation is mandatory in handling heterogeneous and disparate data models. This can be achieved through some fundamental homogenous model. To comply with these varieties of events across many disciplines, we need a generic method of representing events.

In the modern era, one of the great advantages of computer tools is the automatic reading of time. In a digitized world, therefore, an event is a natural representation of information identity with its content, associated context, and time. However, the storage design for such an event is not that simple and has been designed proprietarily to meet various application demands.

Although a database system supports methods to store and retrieve digitized data, an impedance mismatch exists between databases and events. Databases are developed to process bulky structured data sets, whereas events are literally generic objects representing something that happens. In the view of database management, an event would be an index of information objects. The problem in using databases is that an event can be any data type and/or associated with a variety of data schemes. Because database systems were developed for structured information sets, types of structures should be priori known. This makes generic events improper fits for a database system. Thus, this work materializes a generic event as a means to find a way for database processing. The following sections discuss the mathematical formulation of an event as a materialized data object and as the identity of an associated data instance.

3.2 An information identity and a pure event

Within the various meanings of an event, we are interested in its abstract representation power in two aspects: (1) an object representing something that happens and (2) its repetitive occurrence invalidating its status as an event. In other word, an event symbolizes something unique and distinct within the group of organizations. In this context, we review three theories of events from the philosophical literature.

The first theory, Jacques Derrida declared in [40], is:

> *An event itself is a primitive object indicating something that happens. The primary characteristic of an event is that not only it does come about as something unforeseeable, not only does it disrupt the ordinary course of history, but it is also absolutely singular. Repetition without the erasure of the first occurrence will not be an event.*

He argued that an event must be exceptional, an exception to the rule, a requirement by which he attaches singularity to an event. No event exists once there are rules, norms, and hence, criteria to evaluate this or that, what happens, and what does not happen. He explains an event by using binary logic, possible or impossible to predict. The *Maybe* probabilistic operator is necessary to handle partially predictable events. Therefore, eventfulness depends on an experience of its rarity. It could be a weight factor of an event in the query computation.

From the viewpoint of data processing, an information system is a transformer that collects event-associated information and converts it into data objects. In this procedure, an event is a something new and therefore, a unique identity of newly registered data objects. Repeating events will not be duplicated, but the reference to existing instances will be increased as a *Maybe* measurer and referred to as the data frequency.

In the second theory of a pure event, Gaston Bachelard affirmed the abstractness of an event that is independent of the application domain [8]. He argued that the instant at which something happens should be understood as a pure event [9]:

> *When I say that a phenomenon taken as a whole changes from state A to state B, what I mean is that between A and B there are myriad details and accidents which I ignore but which it is always in my power to indicate.*

His viewpoint on the nature of time takes issue specifically with Henri Bergson's notion of duration [16]. For Bachelard, contrary to Bergson, our personal history is neither a memory

of continuity nor does it contain the entirety of our past; instead, it consists of selected memories. Hence, the experience of lived time is fractured, interrupted, not singular and continuous; he argues that there is no underlying thread showing that time is multiple and discrete because our memory does not even give us direct access to the temporal order; it needs to be supported by other ordering principles.

Again in the view of data processing, a pure event (PE) is like a burst of data objects. It may be an infinitesimal object occupying a minimal duration of time. A PE can be understood and interpreted in various ways by associated content and by the state of the surrounding environment, also called context, in which it happens. A PE should be an abstract object representing a generalized identity that is independent of the type of associated data objects or linked relations.

The third theory on events is Jaegwon Kim's definition on events [74]. Kim theorized a structured event, e, which is composed of three entities: an object (x), a property (P), and a time (t),

$$e = [x, P, t]. \tag{14}$$

In his theory, a unique event is defined by two principles:

EP-1 The existence condition: "$[x, P, t]$ just in case substance x has property the P at time t."

EP-2 The identity condition: "$[x, P, t]$ is $[y, Q, \bar{t}]$ just in case $[x = y, P = Q,$ and $t = \bar{t}]$."

It should be noted that Jacques Derrida's ideas [40] on the singularity and discontinuity of events accord with Jaegwon Kim's. The aforementioned three theories from philosophical backgrounds formulate the rules against which in the following sections we formalized our ideas of an event.

3.3 Spatio-temporal aspects of events

This section reviews the fundamental aspects of an event in two categories: space and time. Let us first review the temporal aspect of an event.

Jaegwon Kim's event model in Eq. 14 represents an event in *n*-ary tuple in which a temporal property is an essential predicate of e. He left time *t* in abstract form without specification of the type of *t* data object that affects how data objects are processed in the database system. He assumed that the data registration time always matches with the actual time an event happens. However, people may argue that the time should be the moment when a data object is registered. Both are semantically different, and the time value could be physically different when a temporal delay occurs between the event moment (i.e., real-world time) and the database registration moment (i.e., database system time). In temporal databases, time is differentiated into two classes [72]: transaction time t_t and valid time t_v.

Definition 3.1. A transaction time t_t is a timestamp of the moment when it is newly registered in the database system. Functionally a transaction time t_t is consistent with the serialization order of transactions

Definition 3.2. A valid time t_v of a fact is a moment when the record is true in the modeled reality. t_v could be a predicate of an entity that constitutes an event with the real-world time.

What we mean as an event in this work is based on the database view of the outer world. From the inner database system viewpoint, the transaction timestamp of a data object is the valid time of a newly registered event. Thus, we set the timestamp as the essential property of an event. This keeps the database system independent of application domains, which may be multifarious in the way they handle temporal information. If an information system is not designed for real-time monitoring or does not have enough temporal granularity of user needs, then there exists a time delay between valid time and transaction time. Such valid time data objects in any format or in any meaning related to events are linked to an

external event as a child to its properties.

Until now, we have mentioned only the temporal aspect of an event. One may argue that a temporal entity is the only necessary property of *e*. Some spatial database researchers may suggest the inclusion of a spatial data object because spatial continuity is the general continuity condition in their arena. They might say any discontinuities in the space should be recorded and treated as events.

In response, we think that temporal continuity makes more sense when applied to events. A measure of time can be assumed to be universal with wide acceptance. The condition that specifies time continuity is simple: Two events are contiguous in time if they are temporally overlapped. Spatial continuity, however, makes the most sense when applied to physical objects, especially material bodies; intuitively at least, we surely understand what it is for two bodies to be in contact or to overlap. For events, the very notion of spatial location often becomes fuzzy and indeterminate because one event may be associated with many or even unknown spatial locations.

For instance, Jaegwon Kim questioned one case at [74] when members of a board of directors living at different places joined in a teleconference and elected person *A* living in Philadelphia as the next president. Exactly where did this event happen for *A*? Obviously multiple locations are related with this complex event. This situation may also violate the EP-2 identity condition as well. Thus, restriction of the location of an event to one place does not solve the problem. From the viewpoint of information, this complex election event groups identity without necessarily having a spatial data object at this higher event level but including multiple sub-events associated with locations.

Using the classification method for time, these spatial locations can be considered as *valid* locations, and they are nonmandatory properties of an event. In other words, *transaction* locations are not available in many cases when a database system deploys multiple data sources in disparate types. This prototype database that we developed in this work includes a transaction timestamp as the essential property of an event.

This does not mean that a spatial location cannot be a property of an event. For instance, we could build a database system to log events on personal cell phone usage. Then if a cell phone is equipped with a GPS, all events recorded using it can have both transaction timestamps and geographic locations as the essential properties of events.

3.4 The E-model data objects and predicates

To materialize an event as a data object for processing, its object type and the way to create a relation with others should be clearly defined. This section introduces the fundamental object definitions and their roles in the E-model system.

3.4.1 An e-node for a pure event

Functionally, a property P in Eq. 14 describes x. It may tell x's height, weight, shape, or color. x may have none or many properties. An application may add additional event properties if it holds the identity of an event. Thus, from the viewpoint of the database, x can be understood as a unique grouping identity of the composite dataset P. Both x and P are data objects, and their relations are understood as functions that form directional links from the source x to the properties P. The primary question is then what data types x and P should be. Should they be the same data type so that the processing system can handle them in a unified way? Or should they be heterogeneous, in which case the methods for handling objects are separated for efficiency or speed?

A design criteria applicable to this situation is that all events in a database system have an identical structure composed of one parent grouping node with all leaf property nodes and no leaf node can be inherited by other nodes. In such a case, processing x and P separately would have several advantages in terms of storage space, indexing, and search speed. However, if we want to model a complex event in which each node can be inherited or propagated into other events, then x and P should be identical data types because any of them could be the grouping identity for other events. This work chose the latter approach, which in practice is better suited for modeling real-world complex events

and their associated data objects.

In summary, we need to define two data objects: (1) an atomic data object that can naturally represent both x and P, and (2) a directional relation object to represent the association between atomic objects. A atomic data object is typically composed of two tuples, which is a set of objects for each symbol (or name) and value. We mean a symbol as the name of a data object conveying the semantic interpretation of the associated data value for object identity.

The next decision criteria concerns whether the data type of a symbol and a value should be equal or different. Whether two data objects should be tightly bundled as one set or whether they are independent to be shared by other data objects in the system is also a question. One case from multimedia analysis applications may exemplify this situation.

One image may have tens of objects to tag. When applied to video data, the number of objects and their dynamics would dramatically increase. In this condition, one value may be insufficient to represent the full semantic meanings of data because the size of the data is growing and the data types are gaining in complexity. Also, spatio-temporal or semantic variations exist in multimedia. Hence, one value object may be mapped to multiple symbols. Conversely, one symbol may be mapped to multiple values. To share objects between symbols and values, both should be identical data types. In conclusion, a new object that represents this many-to-many mapping should exist between value objects and symbol objects to make the connection distinct. We define this e-node object as the atomic object of an event.

Definition 3.3. A pure event θ_e named an e-node is the identity of an ordered ternary (3-tuple) set,

$$\theta_e = [\Phi_\delta(\theta_e), \Phi_v(\theta_e), t], \tag{15}$$

where $\Phi_\delta(\theta)$ is a symbol of θ_e, $\Phi_\delta(v)$ is the value of θ_e, and t is a transaction time. An e-node is a persistent and unique object. Within a system no two e-nodes can share the same name and value set.

3.4.2 A group e-node as a generalized event

Linking the concept of an e-node to Kim's model, an e-node can be considered as one stand-alone event with fixed entities. Then a pure event in Kim's model is the subjective association of multiple e-nodes in entity-property relations in which both entity and property objects are all identical type e-nodes. Hence, an e-node for an entity object, which we named an e-group node, is an identifier of associated property e-nodes that exemplify the data object. For Kim's model, θ_g is an event instance.

Definition 3.4. A group e-node or e-group node, θ_g, is a special e-node with a unique identifier of associated child e-nodes.

An e-node embeds a timestamp that logs its transaction time. Thus, relations between e-group nodes should conform to the temporal order that formulates an acyclic graph. This ensures that the length of a path between two e-nodes, if it exists, is bounded by the acyclic path. The acyclicity of the e-group nodes graph is assured by the temporal constraint that the timestamp of a new e-group node should be newer or equal to old e-group nodes:

$$\Phi_t(\theta_{g_i}) \geq \Phi_t(\theta_{g_j}), \forall i \geq j, \tag{16}$$

where Φ_t is the temporal predicate that projects the timestamp of an e-node. Details on the acyclic graph formed by e-group nodes are introduced in Chapter 7.

3.4.3 C-data object

The data type of an e-node symbol and value objects was not clarified in Definition 3.3. This data object is named c-data in our work and represents the composite data that encapsulates disparate data types and provides unified access to the raw data values. C-data is the primary identity of any composite raw level data that an information system can handle. C-data could be a basic integer, or a text, or a blob object like an image, or something else depending on what the application needs. One c-data design case is introduced in Section 3.5, in which the raw data back-end is a set of separated silos for each data type. The

purpose of this separation is to get the maximal support from existing database systems because most of them have data type-specific functions and indexing methods.

3.4.4 E-node relation

To model real-world information, relations between e-nodes should be clearly defined. And the rule of association should be flexible and concrete for database system implementation. In our work, the relation is materialized, not left an abstract object as in other data models. The relation should follow the temporal order of e-nodes by their transaction timestamp. A relation e-node can be interpreted as a functional transformation from the source e-node to the output e-node.

3.4.5 E-model

With all aforementioned objects, the E-model can be formally defined as:

Definition 3.5. The E-model is a linked list of relations, e-nodes and c-data objects. An e-node is a physical instance of a PE and is an identity of two c-data objects: symbol and value. An e-node embeds a transaction timestamp that logs the creation time of the e-node. A relation object is materialized in the E-model as an e-node to represent the directional relation from the source e-node to the target e-node.

Figure 18 graphically depicts the structure of the E-model by using the object-role modeling (ORM) [62, 18, 63] that we used to convey the complete roles and relations between E-model objects with conditional predicates. ORM provides a more efficient graphical model description method than the set representation, which needs additional descriptions of the role and constraints of objects.

T. Halpin provides an excellent tutorial [62] in the details of ORM diagram design. This section briefly introduces the ORM design procedures that are focused on conceptual modeling. ORM simplifies the design process by using natural language information analysis (NIAM) [103] as well as intuitive diagrams that can be populated with instances. This is

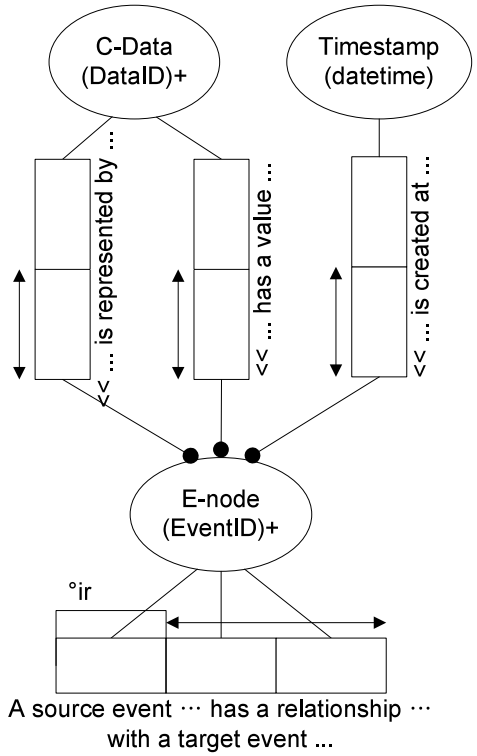


Figure 18: The E-model structure in ORM as a set of c-data, a timestamp, an e-node and e-node relations.

done by examining the information in terms of simple or elementary facts like objects and roles to provide a conceptual approach to modeling. Its first formalization in object-role modeling appeared at [62, 18]. For a list of graphical notations, please consult [63].

In Figure 18, three fact types (e-node, symbol c-data, value c-data, and timestamp) and four predicates exist (three predicates of an e-node and the relation predicate for e-nodes). Entity types are depicted as named ellipses. Reference modes in parentheses under the name are abbreviations for the explicit portrayal of reference types. Predicates that explain the relation between two entity types are shown as named sequences of one or more role (rectangle) boxes. An n-*ary* predicate has n role boxes. For instance, the timestamp of an e-node has the binary predicate with two role boxes. The relation predicate, which needs three e-nodes, has three role boxes.

Predicate names under the role boxes are read left-to-right and top-to-bottom. If they are preceded by ≪, then the reading direction is reversed. If a role is mandatory for an

object type, this is explicitly shown by means of a circled mandatory role dot where the role connects with its object type. In the E-model, three roles of an e-node (symbol, value, and timestamp) are mandatory and must exist to define an e-node.

A bar across a role is a uniqueness constraint that is used to assert that entries in one or more roles occur there once at most. A bar across n roles of a fact type ($n > 0$) indicates that each corresponding n-tuple in the associated fact table is unique (no duplicates are allowed for that column combination). Arrow tips at the ends of the bar are needed if the roles are noncontiguous (otherwise arrow tips are optional). As a result, in the E-model, one e-node can have only one set consisting of a symbol, value, and timestamp.

Definition 3.6. The e-node in the E-model is a three-tuple object in which both a symbol and a value are referred from the c-data set.

$$e = \{d_s, d_v, t\}, \tag{17}$$

where d_s represents a symbol c-data object, d_v is a value c-data object and t is a timestamp with the following properties:

EN-1 An e-node is a connection identity for the set of a symbol and a value.

EN-2 The transaction time log is attached to keep the history.

Using c-data objects for both symbol and value significantly changes the way of handling data objects in the information system. This unique architecture is devised based on the observation that in complex data models, data symbols are objects to query and search. However, most data models treat symbols as metadata and thus, process them separately. Consequently, the query mechanism of these models depends on their proprietary storage architecture. However, in information retrieval for unstructured data objects like texts, the name of data objects can be the target of a query. For instance, a user may want to query all data objects labeled face image and registered today. This illustrates that in an unstructured data model, a symbol and a value are both objects to search. Thus, the E-model uses the

same data object type and identical storage for both symbol and value objects. This permits a query to be simplified and also to interoperate easily in the distributed network in sharing data information.

The relations between e-nodes at the bottom of Figure 18 shows the way e-nodes are related to each other. They are represented in the adjacency list of three tuples in which the relation object is located in the center. All three objects in the relation list are referred to e-nodes in which the type of a center e-node is limited to relation e-nodes. An instance of the e-node relation is a triple set:

$$r_x = \{e_s, e_r, e_t\}, \tag{18}$$

where e_s is a source e-node, e_t is a target e-node and e_r is a relation e-node.

3.5 The c-data storage model

A c-data is designed to achieve independence from types of disparate raw data. Raw data type means those types of data that back-end storage differentiates in the schema definition. In relational databases, VARCHAR, INTEGER, DECIMAL, or BLOB raw data types are such cases. Application developers may define their own data objects for c-data and link them to the e-node. The c-data model shown in Figure 19 is one design case for the sake of illustration. Raw data types in this design represent the data types supported by the database system on which a c-data is implemented.

Definition 3.7. The c-data is a identity over different types of raw data entities. The raw data type of c-data is specified with a *RawDataType* predicate. This design permits one raw data per one c-data object with the following properties:

CD-1 The c-data object behaves as a composite data index over different types of raw data entities. The connection between the *RawData* and C-data is exclusive OR (“⊗ in the diagram”)which means that only one raw data can be mapped to the c-data.

CD-2 The number and type of low-level data connected to the c-data object can be freely designed to meet the demands of the application domain.

CD-3 The c-data architecture is independent of the parent e-nodes and their relations.

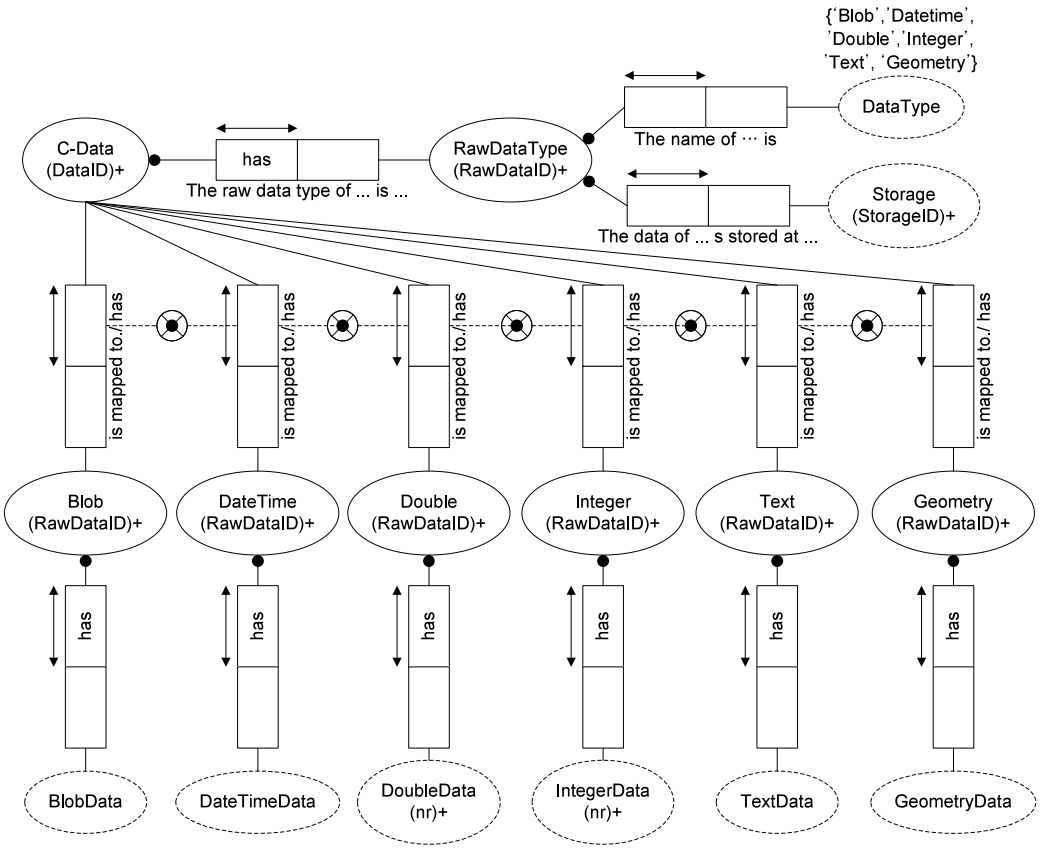


Figure 19: The c-data storage model.

Figure 19 resembles the star schema [80] that is popular in data warehouse design. A star schema is a typical way of implementing a unified data index over different heterogeneous data silos. Details of the c-data storage model are as follows:

CS-1 The raw data type of c-data is specified with a *RawDataType* predicate in Figure 19. ORM represents an exclusion constraint (exclusive-or) by a circled *X*.

CS-2 Figure 19 is normalized not to have *Null*.

CS-3 Value types displayed as broken ellipses mean that they are constant and have no sub-entities.

CS-4 Lines connect object types to the roles they play.

CS-5 It has value constraints that show a list of possible values in the form of an enumeration or range in braces.

CS-6 *Storage ID* is a raw data table reference.

If a *c-data* is successfully registered, an instance of a *c-data* is an ordered list of entities:

$$d = \{\mathcal{I}_d, \Phi_D(d), \Phi_\tau(d)\}, \tag{19}$$

where $\mathcal{I}_d$ is a unique identifier of d, Φ_D is a predicate that projects the raw data type of d, which would be a *Storage ID* of a raw data object, and Φ_τ projects an identifier of an associated raw data object.

CHAPTER IV

SEARCH CONSTRAINTS FOR THE E-MODEL

Search constraints in a query are to select e-nodes of interest and to project their properties. In the E-model, three types of objects exist in hierarchical tiers: (1) *c-data*, (2) *e-node*, and (3) *relation*. Thus, search constraints are functional projections on multiple levels by their hierarchical relations. They can also prioritize the order of e-nodes for user interests or build a probability density model of instances. This chapter describes in details such constraints and the algorithms used in the E-model to determine rank. Section 4.1 describes search constraints for c-data objects. Section 4.2 and Section 4.3 are for e-node objects and relation objects. For information retrieval of a large number of records, the order in which results are output is very important for usage. E-model algorithms to determine rank are developed in Section 4.4 to meet such demands. While ranking is based on popularity, the rarity of an event also plays an important role in finding discontinuities in the event flow. The measure for such rarities is named eventfulness and is introduced in Section 4.5.

4.1 C-data search constraints

C-data objects, where $d = \{\mathcal{I}_d, \Phi_D(d), \Phi_\tau(d)\}$, are abstract concepts to make the E-model system independent from an application domain. Thus, operators for specific raw data types will not be discussed in this work. For instance, geospatial operators like *Contact* or *Overlap* are left for the application designer. Instead, we assume the output of such low-level functions is a set of c-data objects that are an ordered list of the c-data object index: $\bar{\mathcal{I}}_d = \{\mathcal{I}_1, \mathcal{I}_2, ..., \mathcal{I}_M\}$, where M is a positive integer. Search constraints then can be applied for each predicates function Φ_D or Φ_τ or they could be a composite of both.

Definition 4.1. A c-data query function, Q_c, is a selection and projection from a given

search condition to query the c-data set. Q_c returns a set of c-data objets $\bar{\mathcal{I}}_d$ as its output:

$$\bar{\mathcal{I}}_d = Q_c(\Phi_D, \Phi_\tau), \tag{20}$$

where a set of input conditions Φ_D and Φ_τ is the juxtaposition and/or composition of operations on the raw data type and its value. Output $\bar{\mathcal{I}}_d$ should be an ordered set of c-data identities.

Example 4.1. Search "911" for all raw data types:

$$\bar{\mathcal{I}}_d = Q_c(*, \text{“911”}),$$

where the search condition "*" specifies the full range of selectable objects.

Example 4.2. Search "meeting" text data with query synonymous semantic expansion:

$$\bar{\mathcal{I}}_d = Q_c(\text{“varchar|text”}, Synonym(\text{“meeting”})),$$

where the first argument specifies the raw data types in Boolean expression. *varchar* represents variable length characters. *Synonym* is a function that returns a set of similar words.

We assume that two arguments in Q_c acts as the AND condition which means that found *c-data* objects found in $\bar{\mathcal{I}}_d$ should satisfy both conditions.

Lemma 4.1. The computational complexity of a c-data query is $O(m)$ in big-O notation [68], where m is the number of disparate raw data storages that Φ_d will search.

Proof. As described earlier, the values of all c-data objects are stored in fully indexed and normalized raw data silos. This means that the time to access the record at each silo is fixed and can be assumed as constant. Therefore, the query time on c-data depends linearly only on the number of raw data silos to search. □

4.2 E-node search constraints

For an e-node, where $\theta_e = [\Phi_\delta(\theta_e), \Phi_v(\theta_e), t]$, constraints are applied to searching over e-nodes of interest by their two c-data objects for e-node names and e-node values. In addition, temporal windowing can be applied to limit the search space by e-node timestamps. Constraints for each entity of an e-node can be composed inclusively or exclusively.

Definition 4.2. An e-node query function, Q_e, is an e-node selection query from given search conditions to search e-nodes with two sets of c-data search constraints Q_c for e-node symbols and values and Q_t for temporal range selection. Q_e returns an ordered list of e-nodes indexes, $\bar{\mathcal{I}}_e$, as its output:

$$\bar{\mathcal{I}}_e = Q_e(Q_{c_s}, Q_{c_v}, Q_t). \tag{21}$$

In Eq. 21, Q_{c_s} and Q_{c_v} have subscripts to specify its role in finding the symbol and the value of an e-node. They are identical Q_c functions because an e-node refers both to symbol and value from the same c-data storage.

Example 4.3. Search all e-nodes having blob data and were registered yesterday:

$$\bar{\mathcal{I}}_e = Q_e(*, Q_c(\text{“blob”}, *), Day(Now()) - 2 \text{ day} < Day(t) \leq Day(Now()) - 1 \text{ day}).$$

Example 4.4. Search all blob materials named with “meeting” synonymous words and created within the last 12 hours:

$$\begin{aligned}\bar{\mathcal{I}}_e = Q_e(&Q_c(\text{“varchar|text”}, Synonym(\text{“meeting”})),\\ &Q_c(\text{“blob”}, *),\\ &Now() - 12 \text{ hours} \leq t \leq Now()).\end{aligned}$$

Example 4.5. Search an e-node for a speaker “Rachel”:

$$\bar{\mathcal{I}}_{e_1} = Q_e(Q_c(\text{“varchar”}, \text{“speaker”}), Q_c(\text{“varchar”}, \text{“Rachel”}), *).$$

Example 4.6. Search an e-node of an episode ID “601”:

$$\bar{\mathcal{I}}_{e_2} = Q_e(Q_c(\text{“varchar”}, \mathit{Like}(\text{“episode”})), Q_c(\text{“integer”}, 601), *),$$

where *Like* is a function that finds texts that start with a given word.

Lemma 4.2. Finding e-nodes using the temporal range is the fastest query.

Proof. For instance, if the query finds e-nodes from the last one hour, then it does not need Q_{c_s} and Q_{c_v}. In fact computational complexity is a constant if e-node timestamps are indexed. □

Lemma 4.3. The computational complexity of an e-node query is $O(m + n)$, where $m + n$ is the count sum of disparate raw data storages in the query Q_{c_s} and Q_{c_v} .

Proof. For an e-node query, Q_c needs to run two times to find a name and a value of an e-node, whereas Q_t can be assumed to be a constant if it is indexed. Hence based on Lemma. 4.1, the computational complexity is the sum of Q_{c_s} and Q_{c_v}. □

4.3 Relation search constraints

In Figure 18, the E-model relation is composed of three rule boxes in the order of source, relation and target e-nodes. The distinction made in this diagram is that all three objects building the e-node relation object are all identical e-node type objects. Thus, the search constraints for e-node relations are three concatenated Q_e functions.

Definition 4.3. An e-node relation query function, Q_r, is a relation selection query from given search conditions on three entities of a relation set. Q_r returns a set of relations $\bar{\mathcal{I}}_r$ composed of three e-nodes at each row of its output:

$$\bar{\mathcal{I}}_r = Q_r(Q_{e_s}, Q_{e_r}, Q_{e_t}). \tag{22}$$

Example 4.7. Find an e-group node of the speaker "Rachel" e-node and the episode ID "601" e-node. Let us refer e-nodes from Example 4.6 and Example 4.5:

$$\begin{aligned} \bar{\mathcal{I}}_{e_3} &= Q_e(Q_c(\text{“varchar”}, \text{“\_eFunction”}), Q_c(\text{“varchar”}, \text{‘\_eGroup”}), *), \\ \bar{\mathcal{I}}_{r_1} &= Q_r(*, \bar{\mathcal{I}}_{e_3}, \bar{\mathcal{I}}_{e_1}), \\ \bar{\mathcal{I}}_{r_2} &= Q_r(*, \bar{\mathcal{I}}_{e_3}, \bar{\mathcal{I}}_{e_2}), \\ \bar{\mathcal{I}}_{r} &= \bar{\mathcal{I}}_{r_1} \cap \bar{\mathcal{I}}_{r_2}, \end{aligned} \tag{23}$$

where $\bar{\mathcal{I}}_{e_3}$ is an e-function for e-group relation. $\bar{\mathcal{I}}_{r}$ is an intersection of two relation e-nodes. It finds common e-group nodes that have both $\bar{\mathcal{I}}_{e_1}$ and $\bar{\mathcal{I}}_{e_2}$ as its child e-nodes where both are connected in $\bar{\mathcal{I}}_{e_3}$ relation from the source e-node to the target e-node.

In the E-model, relations are finite and unique (See Definition 3.4 PE4). This limits the order of complexity of connected object graphs. In the view of database implementation, this ordered list behaves as an adjacency list representing parent-child relations between e-nodes.

Lemma 4.4. The computational complexity of an e-node relation query is $O(s + t)$, where $s + t$ is each the sum of disparate raw data storages in the source e-node query Q_{e_s} and the target e-node query Q_{e_t} .

Proof. There are three Q_e functions in the e-node relation query, and by Lemma. 4.3 the computation complexity would be the sum of these three complexities. However, the relations in the E-model are finite, and the name of relations is fixed as "*_eFunction*." Thus, the memory cost to store these relation e-node indexes is negligible. Because running relation e-node queries for every e-node relation queries slows performance, the e-node relations are cached at the run-time before the query. Therefore, the computational complexity only counts the source and target e-node queries. In practice, queries on e-node relations are finding the source or target e-nodes in some fixed relation from an give source e-node. This means that in most cases Q_e will run only one time. Thus, the burden added to that of

e-node queries lies in searching the relation table. If an e-node relation table is properly indexed, then the computation time becomes a fixed constant. □

4.4 E-nodes ranking algorithm

In information retrieval, when a query does not specify the exact category to search, the output may include a variety of heterogeneous data objects associated with different data models. Moreover, they may contain millions or more sets; as a result, an output algorithm that sorts the results and ranks the most relevant ones at the top is very necessary for users.

In set-based data models, the order of the output depends on the data value. For instance, in SQL, "ORDER BY *variable* DESC" in a SELECT statement sorts an output set by the value of *variable* in descending order. In the case of the E-model, we mentioned that all data objects in the E-model are fully normalized. This means that no redundant data objects exist and that data objects exist in the first-normal form (1NF) [73] , which does not allow *Null* objects. Binding e-node relations with first-normalized data objects naturally creates a reference count that can be used to order query results. We mean the reference count of an e-node by its linked count in the e-node relation. For instance, when Q_e returns multiple e-nodes, they can be ordered by their reference count.

The physical interpretation of the reference count varies by application. If relations represent a family pedigree, then the count directly means the number of children. If relations are for text information retrieval, the count may mean word frequency. In complex application domains, this count may refer to the probability of selected e-node occurrence.

The granularity of data values is the other perspective to consider in reference counts. Granularity may span across its application to all raw data types. For texts, a query word can be expanded from its base form to its derivatives or to synsets in various relations. For geographical objects, tolerances in matching areas, angles, or shapes may work as granularity. For numbers, the order of granularity would be value ranges or number precision.

For date-time values, seconds, minutes, hours, or days would be such a unit for granularity. In information retrieval such as a keyword search, the query result on a corpus or pages may return millions or more records. Then the sorting algorithm needs some priority model to reorder the output to satisfy user interests. Most Internet search engines have their own priority rank-ordering model [23, 136, 22, 59, 94], and the main feature used in such a calculation is the reference count of a Web page within a group of pages. This granularity concept can be naturally applied to the e-node query and should be supported in the reference count.

Definition 4.4. Let us formulate the E-model reference count as:

$$P_e = f_n(e \in E)/f_n(E) = f_n(Q_e)/f_n(E), \tag{24}$$
$$0 \leq P_e \leq 1, \sum_{e \in E} P_e = 1,$$

where e is a set of interested e-nodes, E is a whole set of e-nodes, f_n calculates the total member and P_e is the probability of an e-node.

P_e in the E-model can be calculated deterministically from the E-model relation table. Specifically the reference count of an e-node mapped in the e-node relation table can be calculated from forward or backward references by the direction between the source and the target e-node and their relations. Also P_e can be calculated with various constraints in the following sections.

4.4.1 Category constraints

In the E-model, pure e-nodes are shared by distinct e-group nodes that are instances of e-categories (See Definition 5.3). Thus a probability computation can be limited to instances of a specific e-category:

$$P_{e_c} = f_n(e \in E_C)/f_n(E_C) = f_n(Q_e)/f_n(E_C),$$
$$C = \{c_1, c_2, ..., c_n\}, \tag{25}$$

where C is a set of interested e-categories and E_c is a set of e-category instances in which multiple e-categories can be specified. With this constraint, e-nodes are children of e-group nodes of e-categories and their structured relations can be back-calculated from the e-node relation table. Let us specify Eq. 25 in detail.

$$E_C = Q_e(Q_C(\text{“varchar”, “\_eCategory”})),$$
$$E_R = Q_r(e_c,$$
$$Q_e(Q_C(\text{“varchar”, “\_eCategory”}),$$
$$Q_C(\text{“varchar”, “\_eGroup”}), *), \tag{26}$$

where e_c is an e-category ID.

4.4.2 Relation constraints

The E-model supports rich relations between e-nodes. When walking through a graph in these relations, constraints can be applied to confine links to specific relations:

$$P_{e_r} = f_n(e \in_R E) / f_n(E),$$
$$R = \{r_1, r_2, ..., r_n\}, \tag{27}$$

where $\in_R$ limits the search over related e-nodes with R relations and R is a set of interested e-functions .

4.4.3 Temporal constraints

Temporal range selection plays a central role in searching multimedia, temporal databases, and most transactional databases. E-nodes in the E-model inherently embed transaction timestamps. Temporal constraints are therefore applied to E_g to build P_e limited to selected time ranges:

$$P_{e_t} = f_n(e \in E_{t \in T}) / f_n(E_{t \in T}), \tag{28}$$

where $t \in T$ limits the search over the specified temporal region T. In the E-model, e-group nodes are distinct and are not shared as pure e-nodes are. Thus, for real-time applications,

which need to search only recent instances, the targets are confined to e-group nodes:

$$P_{e_t} = f_n(e \in_{R_c} E^g_{t \in T}) / f_n(E^g_{t \in T}), \tag{29}$$

where E^g limits the set to e-group nodes and $\in_{R_c}$ searches relation parent e-group nodes of e.

4.5 A measurer of eventfulness

Jacques Derrida mentioned that eventfulness depends on an experience of its rarity. In this context, eventfulness could be formulated as a weighting factor of an event in query computation. The E-model supports two types of measurers:

$$E_f = \frac{Now() - t_{max}}{Now() - t} \times (1 - P_{e_c}) \tag{30}$$

and

$$E_f = (1 - P_{e_c}) \times e^{-(Now() - t)}, \tag{31}$$

where P_{e_c} from Eq. 25 is the category constraint. Eq. 30 is proportional to the inverse of the time difference, whereas Eq. 31 shows a logarithmic proportion to time difference.

CHAPTER V

E-MODEL SCHEMA OBJECTS

A schema is a structure of data objects. E-model schema objects are composed of fundamental elements for modeling an application domain. Because we are designing a data model and its storage system, data types, their entity sets, and their relations with other objects should be clearly defined. This chapter deals with the fundamental schema objects for the E-model that were introduced in Chapter 3. In all object notations, a prefix *e* is attached to denote that they are inherited from an e-node.

5.1 E-model SD type

In the E-model, an e-node is an ordered 3-tuple composed of its name, value, and timestamp. Because the timestamp of an e-node depends on the transaction time that will be automatically logged when an e-node is registered, the name and the value of an e-node are the specification for an e-node. Consequently, in defining the structure of an e-node, its name and raw value data type are elements of structure specification.

Definition 5.1. An E-model symbol-data type or e-sdtype, e_{sd}, is a type definition of an e-node. An e-sdtype is a set of two predicates: one e-node that stores its name $\Phi_{\delta}(e_{sd})$ and the other e-node that stores its raw data type $\Phi_{\tau}(e_{sd})$. These two e-nodes are connected to an instance of e_{sd} by an e-sdtype entity relation, "*_eSDType_entity*".

Figure 20 explains the structure of e-sdtype in ORM. The *RawDataType* depends on the needs of the application domain. The e-sdtype registration is complete when both *Name* and *RawDataType* nodes are registered as its entities. *EventID* denotes the ID of an e-node instance. This e-sdtype stores the data name and its raw data type (not value). It is an interface for an e-node registration.

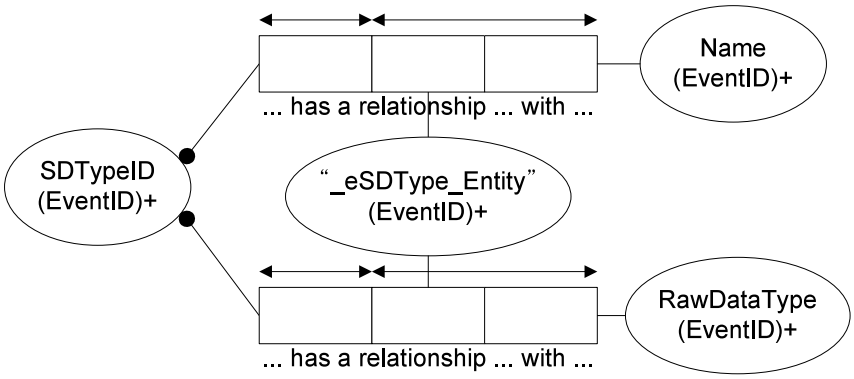


Figure 20: The E-model SD type schema.

5.2 E-model function

An e-node relation introduced in Section 3.4 is a labeled edge between two e-nodes. It represents the semantic relation between two objects. As a result, a schema of an e-node relation has source and target e_{sd} objects. In the E-model, the term *function* is used in place of relation when it is necessary to distinguish the directed relation from the source to the target. The other thing to note is that the E-model is based on an e-node that inherently supports the temporal aspect of information history. So the connection between two e-nodes always has transaction timestamps for comparison of their temporal order. Specifically, for e-group nodes we created a rule that their traverse should follow the temporal order from the old e-group node to the new e-group node.

Definition 5.2. An E-model function or e-function, e_f, consists of a domain (source) e-node X and a codomain (target) e-node Y with constraints on $x \xrightarrow{e_f} f(x)$, where $x \in X$, $f(x) \in Y$ and both are e-sdtypes.

Because the relation object in the E-model is distinct and unique, an e-function depicted in Figure 21 directly uses its name as its identity. An e_f may have input and/or output e_{sd} objects. For each *Input* and *Output* e_{sd}, multiple e_{sd} can be attached which means that when registering e-function instances, multiple e_{sd} can share the same relation e_f.

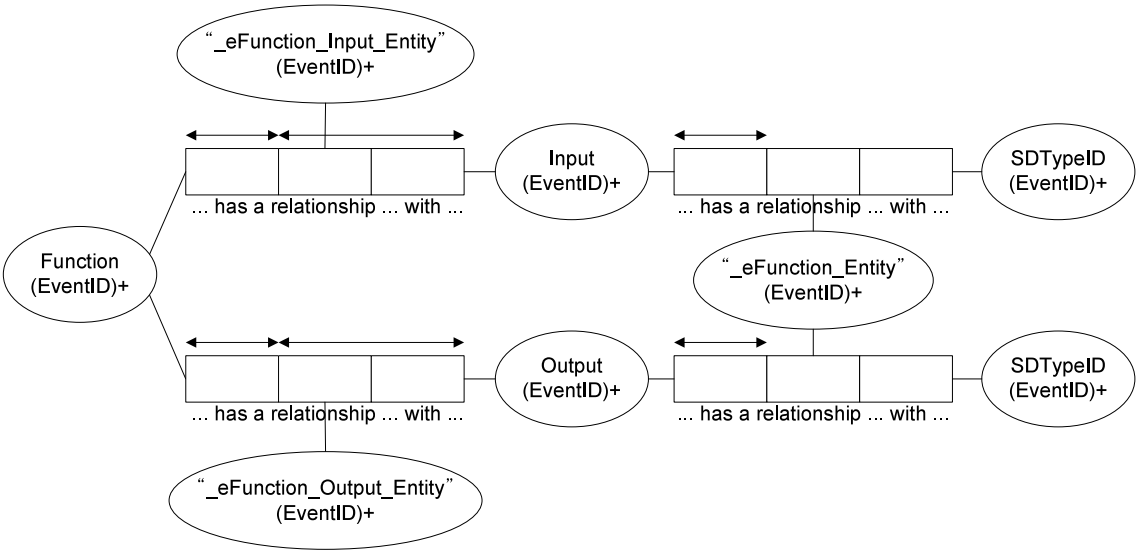


Figure 21: The E-model function schema.

5.3 E-model category

A category of the data model is in general a collection of objects, a collection of functions between objects and child category objects for inheritance. The E-model supports a category in this context in which the aforementioned E-model objects are associated.

Definition 5.3. An e-category, e_c, is a grouping schema object that constitutes a set of structured data objects. An instance of an e_c schema object is an e-group node and its child e-nodes have structured relations with an e-group node. An E-model category consists of three elements: (1) e_{sd} as its basic data objects, (2) e_f to specify relations between data objects, and (3) child e_c sets for inheritance.

The structure of e_c is depicted in Figure 22. The *CategoryElement*, which is in *_eCategory_Entity* relation with *CategoryID*, is an abstract object for three types: *e-category*, *e-function* and *e-sdtype*. Using different objects for an abstract object in the E-model is feasible because all object types are homogenous e-nodes; two *e-nodes* connected by *_eCategory_Entity* relation are *e-nodes*. Our design permits an e-category to include other e-categories as its children by which hierarchical inheritance or nested categories can be

modeled. In doing so, self-referencing is not allowed and this restriction is indicated with an °*ir* specifier in Figure 22.

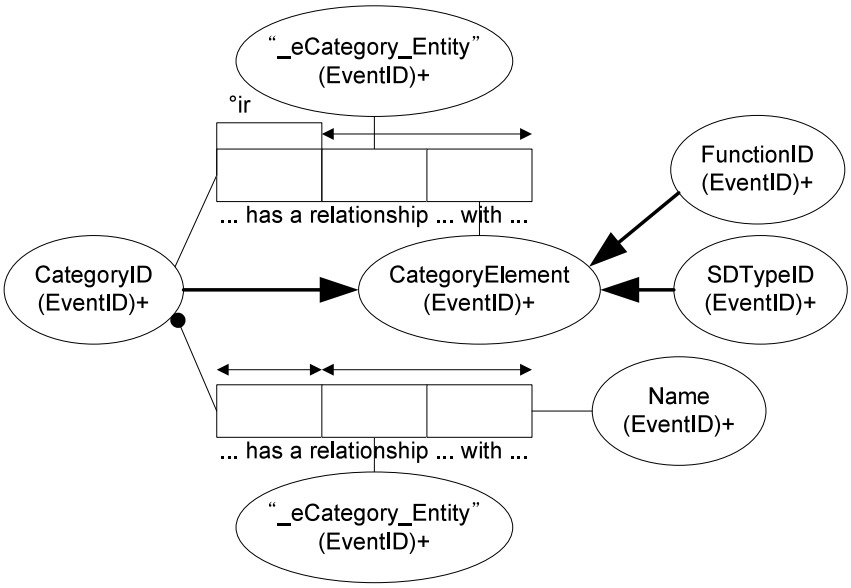


Figure 22: The E-model category storage model.

It should be noted that all E-model schema objects are distinct. This means that any e-categories that include the same e-sdtype or e-function also shares the same e-nodes in their instances. This is equivalent with the *NATURAL JOIN* of relational theory which are all combinations of tuples in each of two tables that are equal on their common attribute names. Thus in the E-model objects can be connected in two ways: (1) by using a directional e-function to connect e-sdtypes and (2) by sharing the same e-sdtype.

Table 2 lists all E-model schema objects and their entities for a summary. It is prepared in a set representation for database design.

Table 2: The list of E-model objects

Object	Tuple	Explanation
Instance objects		
c-data	{c-data id, raw data type, raw data id}	C-data provides type-free access to disparate raw data silos.
tag	{c-data id, word id, lexical id, synset id}	*Tag* is a semantic index of a shared dictionary of a c-data object.
e-node	{e-node id, symbol c-data id, value c-data id}	E-node is an instance of an event composed of two c-data IDs each for the e-node symbol and value.
e-node relation	{source e-node id, relation e-node id, target e-node id}	The relation table represents a directed labeled link from the source e-node to the target e-node.
Schema objects (All e-nodes in tuples are e-nodes)		
e-sdtype	{e-sdtype id, symbol id, raw data type id}	E-node type definition. An instance of an e-sdtype is an e-node.
e-function	{e-function name}	An e-function is a distinct relation that a text name distinguishes itself.
e-function entity relation	{e-function id, *"_eFunction_Input(Output)_Entity"*, e-function entity id}	An e-function is a directed relation from the source to the target. It may have input and/or output entity objects.
e-function child entity relation	{e-function entity id, *"_eFunction_Entity"*, e-sdtype id}	For each e-function input or output entity, multiple e-sdtypes can be attached.
e-category	{e-category id}	Its type is one of the basic, composite, or category of categories.
e-category entity relation	{e-category id, *"_eCategory_Entity"*, e-category element id}	E-model permits e-sdtypes, e-functions and e-categories as its child elements.

CHAPTER VI

E-MODEL DATA MANIPULATION

Data manipulation functions are interfaces for interaction with the E-model system. Basic functions are a set of three operations: (1) insert, (2) delete, and (3) update. The fundamental data object of the E-model is an e-node. The name and value of an e-node are subjective in that these are determined by its associated c-data objects. This subjectivity affects the job order in the manipulation of data objects. For instance, when registering an e-node, its two c-data objects should be preregistered before insertion of an e-node. When deleting an e-node, the order should be reversed. For event relation registration, all three e-nodes should be preregistered. And before that, all c-data objects associated with participating e-nodes should be registered. In the following sections, details of the algorithms are discussed.

6.1 Insert objects

To register a new e-node, the c-data objects of the e-node need to be priori registered. Because the E-model is an archive engine that is normalized to share common data objects, we need to check a priori existing data objects at each data registration. This applies equally to e-node objects and c-data objects but not to e-group nodes, which are instances of an e-category that records the occurrence of events. Thus, even if all child e-nodes are duplicates, an e-group, which is the parent of all child e-nodes, should be registered with a transaction timestamp to record its history of occurrences. To create a distinct e-group node, the e-group node value could be a unique identifier like UUID. Let us first see the insert algorithm for the three basic elements of the E-model. Algorithm 1 describes the c-data insert algorithm. Assuming an abstract interface to the back-end raw data storage, two arguments specify the raw data type Φ_τ and the value of a c-data object Φ_v. In a normalized data model, data registration should check the existence of duplicated instances to

Algorithm 1: Inserting a c-node: *InsertCData*

Data: Raw data type Φ_τ.
Data: Raw data value Φ_v.
Result: *Raw data* type index $\mathcal{R}_i$.
Result: *C-data* index $\mathcal{I}_d$.
begin
REM Find matching *c-data* if exists.
$\mathcal{I}_d \longleftarrow GetCData(\Phi_\tau, \Phi_v)$
if $\mathcal{I}_d == null$ **then**
REM Register *raw data*.
$\mathcal{R}_i \longleftarrow PutRawData(\Phi_\tau, \Phi_v)$
REM Register *c-data*.
$\mathcal{I}_d \longleftarrow PutCData(\Phi_\tau, \mathcal{R}_i)$
end

maintain the uniqueness of data and ensure that the output is a unique index of a c-data object. For an e-node in Algorithm 2, two c-data objects for an e-node name and value should be checked prior to registration. The subscript of $\mathcal{I}$ represents symbol (s) or value (d).

Algorithm 2: Inserting a *e-node*: *InsertENode*

Data: Symbol $\mathcal{I}_d^s \longleftarrow PutCData(R_t^s, \mathcal{I}_d^s)$.
Data: Data $\mathcal{I}_d^d \longleftarrow PutCData(R_t^d, \mathcal{I}_d^d)$.
Result: *e-node* index $\mathcal{I}_e$.
begin
REM Find matching *e-node* if exists.
$\mathcal{I}_e \longleftarrow GetENode(\mathcal{I}_d^s, \mathcal{I}_d^d)$
if $\mathcal{I}_e == null$ **then**
REM Register symbol *c-data*.
$\mathcal{I}_e \longleftarrow PutENode(\mathcal{I}_d^s, \mathcal{I}_d^d)$
end

In the case of e-node relation insertion described in Algorithm 3, two cases exist according to the type of e-node. The first one is when both e-nodes are e-group nodes. This means that both e-nodes are instances of e-categories. Their relation may mean a structured relation and also represent the flow of information in some causal relation. Hence,

we check the temporal order by their transaction timestamp to conform to the RDAG structure. If neither e-node is an e-group but has a specific relation, then their relation is directly recorded.

Algorithm 3: Inserting relations: *InsertENodeRelation*

```
    Data: Source e-node I_e^s.
    Data: Target e-node I_e^t.
    Data: Relation e-node I_e^r.
    begin
REM     Check I_e^r type.
        if Φ_δ(I_e^s) == "e-group" AND Φ_δ(I_e^t) == "e-group" then
REM         Check temporal acyclicity {I_e^s, I_e^r, I_e^t}:
            if {Φ_t(I_e^s) ≤ Φ_t(I_e^t)} then
                PutRelation(I_e^s, I_e^r, I_e^t).
        else
REM         Check duplication
            if GetRelatedENode(I_e^s, I_e^r, I_e^t) == null then
                PutRelation(I_e^s, I_e^r, I_e^t).
    end
```

6.2 Delete objects

For a normalized data model like the E-model, deleting objects should check other objects that reference to them. This applies equally to all object types in the E-model. This section discusses all cases of the delete operation.

6.2.1 Delete an e-node object

Because an e-node can be shared in the E-model, several conditions should be checked in deleting an e-node.

DE-1 To keep a transaction safe, if an e-node to be deleted is a leaf node, the sequence of the deletion should be that associated relations with parent e-group nodes are dropped first and then the e-node.

DE-2 If an e-node is an e-group node (or not a leaf node), deletion should take into account whether to orphan the associated child e-nodes. For instance in Figure 23, if we

want to delete the "B" e-node, then dropping the relation between the "A" and "B" e-nodes is simple. However, whether the child e-nodes should be dropped is not clear. This is a sort of design constraint that depends on the application domain. In a relational database, when the relation called *FOREIGN KEY* between two tables is created, such constraints can be specified at the time of design time [42]. The *ON DELETE (UPDATE) CASCADE* constraint will delete associated parent-child records. The *ON DELETE (UPDATE) RESTRICT* constraint will reject modification of the linked records. For instance , in Figure 23, when a user specifies the delete path ($A \xrightarrow{R_1} B \xrightarrow{R_2} C \xrightarrow{R_3} 2$), then the *CASCADE* condition will delete all e-nodes in the graph even if they are not in the path, whereas the *RESTRICT* constraints will delete none of them.

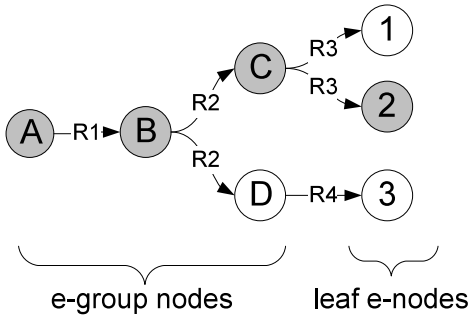


Figure 23: Linked e-node relation graph example.

DE-3 To minimize the loss of graph consistency in the *CASCADE* deletion, we introduced a new *PATH* constraint. With the *PATH* constraint, we first check all e-nodes and relations in the path and then delete all e-nodes that have fewer than one child e-nodes. All relations in the path will be deleted. Finally, e-nodes "B, C" will be replaced with new e-nodes. "A, 1" are the e-nodes to be deleted. E-nodes and relations that survive with the *PATH* constraint are depicted in Figure 24. All relations R_1, R_2, R_3 are dropped.

The *PATH* constraint is formalized in Algorithm 4. One e-node deletion described in DC-1 is the case of a single e-node with no path. In the path, we assume that a path includes a

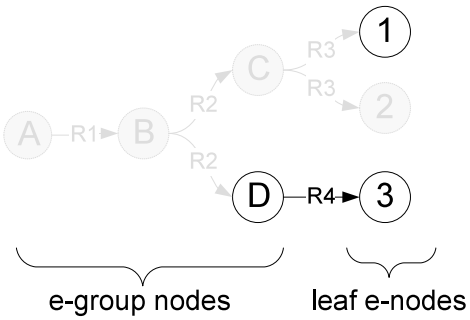


Figure 24: Delete e-nodes with *PATH* constraint.

series of 1-to-1 sequences. Multiple paths like 1-to-N will be treated as expansion of the 1-to-1 path.

Algorithm 4: Delete an e-node path (PATH constraint): *DeleteENodePath*

```
    Data: Source e-node path P_e
    Data: Parent e-node I_e^p
    Data: Relation e-node I_e^r
    Data: Child e-node I_e^c
    begin
REM     Walk the path P_e by one depth.
        while {I_e^p, I_e^r, I_e^c}, = GetNextPath(P_e) do
            if GetChildCount(I_e^p, I_e^r, *) ≤ 1 then
REM             Drop the relation with all child e-nodes.
                DropENodeRelation(I_e^p, I_e^r, *)
REM             Drop the parent e-node
                DeleteENode(I_e^p)
    end
```

6.2.2 Delete a c-data object

In the E-model, c-data objects are abstract encapsulations of back-end disparate data silos, and they are also normalized to be shared by e-nodes. Hence, deleting c-data objects works in a similar pattern as deleting e-nodes, except that c-data is not directly related with other c-data objects. This means that when checking the shared c-data objects, we only need to check whether a c-data is referenced by other e-nodes.

In deleting a c-data object, a user may set constraints between e-nodes and c-data objects in delete or update operations as to whether to delete or update c-data objects in the

CASCADE, *RESTRICT*, or *PATH* modes. One difference is that because a c-data object is used both as the name and value of an e-node, finding the shared objects of e-nodes should look at both aspects. Also, physical removal of c-data objects depends on the application design.

6.2.3 Delete an E-model schema object

Chapter 5 introduced three E-model schema objects - e-sdtype, e-function, and e-category. Before we delete a schema object, we should consider the dependency between schema objects. As noted earlier, each schema object is an independent e-node that can be shared with other schema objects. As a result, the rule of deletion defined in previous sections works equally well for E-model schema e-nodes. However, deleting a schema object means the structure of an associated e-category is modified. Therefore, the associated instances should be dropped to keep the schema consistent. This section describes this part in detail.

- e-sdtype: When an e-sdtype is deleted, its parent e-categories should be checked because all instances of the specific e-sdtype need to be removed. This works like the *ALTER TABLE DROP COLUMN* in a relational database but the scope is not limited to one table but to the database system. Hence it works like the *ALTER DATABASE DROP COLUMN*.

- e-function: A relation e-node, also called an e-function, is a special e-node that represents the semantic relation between e-nodes. It is the E-model schema object (See Section 5.2) and located at the center of the adjacency list that stores the relations between e-nodes. In graph path representation, a relation e-node is the label of an edge that connects the source to the target e-nodes like "*R*" in Figure 23. In the E-model, relation e-nodes are assumed to be finite and predefined to be shared and interpretable in between groups of interest.

  In practice, one relation e-node connects numerous e-nodes because any e-category schema may use the same relation e-node for all its instances. Consequently, deleting

a relation e-node affects associated e-category schema objects and thus, invalidates their instances. If an e-function is about an e-category instance, then deleting a relation e-node removes all instances of an e-category just as the *TRUNCATE TABLE* action does in a relational database. The difference is that the E-model can truncate any e-categories at the same time by deleting the common relation e-node and deleting its instances from the relation adjacency list.

- e-category: Deleting an e-category is the sequence of deleting e-sdtypes and e-functions. It is like *DROP TABLE* in a relational database.

6.3 Update objects

The update operation is the sequence of insert and delete operation. Because both operations have already been discussed in previous sections, no further explanation will be provided except the transaction safety part that is needed to maintain data consistency.

Data manipulation requires considerations of transaction safety. By safety, we mean that in multi-user and multi-session distributed database systems, the state of data should keep the concurrency under data manipulation. For instance, when one user is updating linked child e-nodes of an e-category, another user who accesses the child e-nodes of the same e-category should not be able to retrieve child e-nodes that are undergoing an update. The other user should only be able to read the child e-nodes before or after their update. Therefore, if the back-end storage of the E-model is developed on top of an existing database such as a relational database system, we must properly widen the commit range of the transaction until the update operation is completed. Problems related to this kind of transaction are also known as ACID (Atomicity, Consistency, Isolation, and Durability) [54], which are transactional properties of a database system.

As an explanation, let us borrow terms used in relational database transactions. Relational databases set the transaction range with the use of three commands: (1) *START TRANSACTION*, (2) *COMMIT*, and (3) *ROLLBACK*. *START TRANSACTION* initiates the

transaction and buffers the results of each command executed during the transaction. *COMMIT* updates databases, which instantly changes the state of the database. *ROLLBACK* deletes the buffered results and cancels the update when the final state is not what a user expected it to be and he or she does not want to update.

Similar concepts can be applied to the E-model. Note that the transaction at the raw data level regarding disparate data silos will be left for developers; In fact, raw level transactions are supported by the database system selected as the E-model back-end system. This section focuses on high-level E-model transaction safety.

E-model has a network-type graph in which multiple e-nodes are linked by sharing their names and values through limited relations. In this complex network, instances of one e-category form a structured graph that is typically in a tree-type hierarchical structure. When inserting an instance of an e-category, transactions to insert associated child e-nodes should be concurrent with a parent e-node insertion. That is to say, the *COMMIT* command should occur at the end of all transactions. To facilitate this procedure, when we insert an instance of an e-category, which is an e-group, it should be the last e-node to insert within the transaction. This means something more than merely ordering the transaction. Because by registering the e-group node at the end of the transaction, all associated newly registered e-nodes of an instance will have an identical transaction timestamp that facilitates the grouping of e-nodes by temporal order. Thus, to design a safe E-model system transaction, the *COMMIT* commands needs to be executed at the end of e-node relations registration.

When registering a complex e-category, transaction timestamp matching is important. Let us explain with one example in which we receive a user memo and then parse the sentence using the natural language tokenizer and finally save a parsed sentence into the E-model system. The parsed graph typically forms a constituent tree that looks like a nested tree [87]. The problem is then that even one paragraph with several sentences can be parsed into a considerable tree in which many words, their parts, dictionary matches, relations with other words, and relations with other sentences are related with each other.

Thus their sentence order and user input context data should be inserted into the E-model database to complete the registration.

In doing this, depending on the system performance, the transaction times of each e-node could differ if they are committed separately. But if the e-group nodes of an e-category instance are temporally matched in the transaction, the search time could be significantly reduced. For instance, assuming that we found one e-node of interest by search-by-value algorithm (Details on this algorithm are introduced in Chapter 8) and want to find related e-nodes not only by their relational connection but also by their transaction time, a temporal search directly comparing indexed timestamps would significantly reduce the search range. Thus, in real-time monitoring applications for complex data models, this kind of temporal synchronization of combined search-by-time and search-by-value methods is the key feature for performance enhancement.

6.4 Schema versioning control

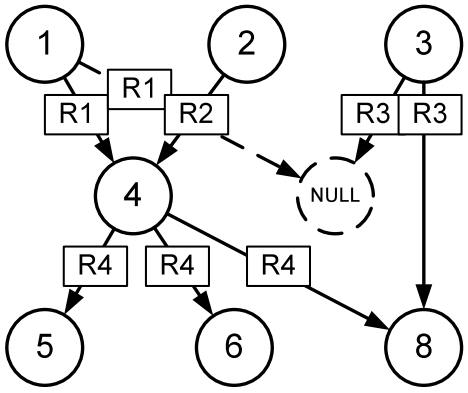


Figure 25: Data schema and instance mismatch problem.

Schema versioning [118], which is also called schema evolution [115] or schema morphism [50], represents intelligent handling of any temporal mismatch between data instances and their data structures. Mismatches occur at incomplete instances of a data schema. For example, when a new schema object is added to the data schema, old instances of an e-category would be incompatible with the new schema. If an existing schema object

is deleted from the schema, then old instances would have abundant entities. This mismatch is in fact a problem of any data schema that evolve over time to incorporate changes in their application domain data model.

Modifying database schema after implementation is in practice a frequent and often troublesome maintenance problem in database administration. Since the birth of data management systems, there have been significant concerns about the schema versioning problem on the part of both database system manufacturers and information system users. B.P. Lientz states in [96] that 50% or more of programmer effort arises as a result of such system modifications after implementation. One of the demands driven by the user community is that schema modification should be as symmetrical as possible so that not only can existing data be viewed through the new schema definitions but also so that data recorded after the modifications can be viewed under the earlier schemata. This is called schema versioning in which data definition functions operate losslessly [118].

In the E-model, because of its fully normalized structure and its time-sensitive nature, the *Null* e-node as in Figure 25 does not physically exist. Thus, a null e-node indicates the missing child e-node of an e-category instance.

Definition 6.1. A *null* e-node is a missing entity of an e-group in which the count of child e-nodes $N(\mathcal{I}_c^g)$ is less than the e-sdtypes of an associated e-category $N(\mathcal{I}_{sd}^c)$. It represents the mismatch between e-sdtypes of an e-category and e-nodes of its instance e-group.

The existence of the null e-node thus can be verified by Eq. 32 without a value examination. Assuming that we have one e-group node e_g to check its validity, the count of child e-nodes of e_g is compared with the e-sdtype count of an associated e-category. If there are missing e-nodes, then there must be null e-nodes.

$$f_N(GetChildEnodes(e_g)) < f_N(GetSDTypeChildEnodes(GetCategory(e_g)), \qquad (32)$$

where f_N is an element counting function.

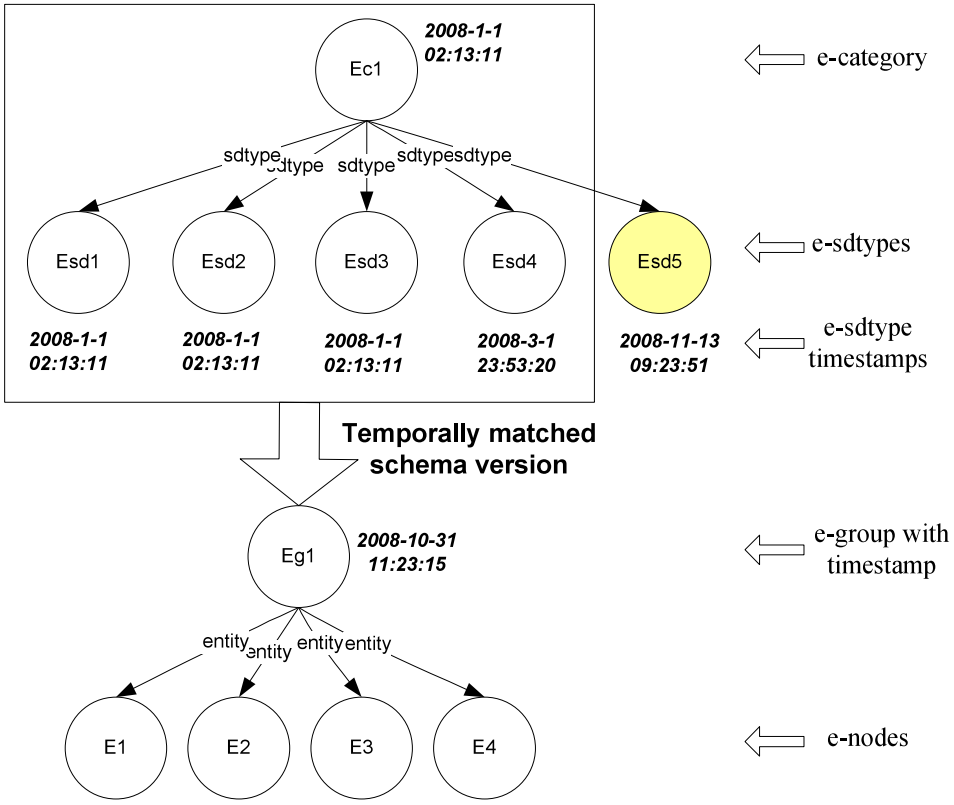


Figure 26: The E-model schema versioning support.

To prevent null e-nodes, the query should use transactional timestamps when it retrieves an associated category of a group instance, which is *GetCategory* in Eq. 32. If e_g is an instance of an old e-category, then *GetCategory* should retrieve the old e-category that may match with e_g which is precisely the schema versioning control. The E-model, because of its inherent temporal nature of events, can support this without loss of information.

As described in Section 3.4, e-sdtype objects in the E-model are identical e-node objects. This means that an e-sdtype node has a transaction timestamp that logs its creation time. Consequently, when e-sdtypes of an e-category are modified or updated, their revision time is recorded at its transaction timestamp. Figure 26 exemplifies this case, illustrating that with a given e-group node temporally matching the e-sdtypes of an e-category are four old e-sdtype e-nodes that were created before the given e-group instance E_{g_1}. This ensures

child e-sdtypes of an e-category temporally accord with their instances. Thus, the temporal correlation property of the E-model can naturally support schema versioning without additional computing overhead.

CHAPTER VII

RELATIONAL DIRECTED ACYCLIC GRAPH

This chapter describes algorithms and rules that govern how all instance objects in the E-model are interweaved to form a network-type graph. Graph theory [114] is often employed to explain the structure of various data models. This chapter illustrates features of the E-model and compares them with existing data models from the viewpoint of graph theory.

In Section 7.1, we introduce the fundamental terms in graph theory. Section 7.2 reviews popular data models and uses their features for structural comparisons. An event may have multiple entities that describe itself. Hence, a group index that stands for an in-stance of an event is necessary. An e-group node for the E-model is introduced in Section 7.3. Section 7.4 describes the relational directed acyclic graph (RDAG) mentioned earlier that is a graph formed of e-group nodes. Because of its acyclic nature, an RDAG may have one or multiple root e-nodes. These root e-nodes are named super e-nodes, which are the root of all offspring child e-nodes. Their properties are discussed in Section 7.5. Implementation of a graph as the storage system requires consideration of its various aspects such as operation, efficiency, and performance. These topics are handled in Section 7.6. In the normalized E-model, e-nodes are shared by e-groups, and each e-node shares c-data objects. Thus, connections between e-categories find such shared e-nodes or c-data objects. A search over search-connected objects forms a M-shaped walk. Detailed constraints and algorithms for this graph search method are formulated in Section 7.7.

7.1 Terminologies

A graph is a collection of vertices connected by links that are also called edges. It can be represented by the vertices in a plane. Between the vertices, a line from $s(e)$ to $r(e)$ can be drawn as an edge $e \in E^1$. When necessary, the edge will be labeled to represent its

role or meaning. A *directed graph*, $E = (E^0, E^1, r, s)$, has directed links that consist of two countable sets E^0, E^1 and functions $r, s : E^1 \to E^0$. The elements of E^0 are called *vertices* (or *nodes*) and the elements of E^1 are called *edges* (or *relations*). For each edge, e, $s(e)$ is the *source* of e and $r(e)$ the *range* of e; if $s(e) = v$ and $r(e) = w$, then we also say that v *emits* e and that w *receives* e, or that e is an edge from v to w (See Figure 27 and [11] for graph theory terminologies).

$$v \xrightarrow{\quad r \quad} w$$

Figure 27: Basic graph notation.

For instance assuming $E^0 = \{e_1, e_2, e_3\}$ and $E^1 = \{r_1, r_2\}$, a graph can be depicted as below.

$$e_1 \xrightarrow{r_1} e_2 \xrightarrow{r_2} e_3$$

Figure 28: The linked list of first-order relations.

Directed graphs have been applied in many applications. A city street map, abstract representations of computer programs and network flows are cases that can only be represented by directed graphs. Directed graphs are also used in the study of sequential machines and in system analysis in control theory. In this work, we are more focused on their power to represent the directional flow of information. With the acyclicity of the graph, a directed graph becomes a directed acyclic graph (DAG) with no directed cycles; that is, for any vertex v, there is no nonempty directed path that is a path that includes at least one vertex and starts and ends on v as in Figure 30. The number of arcs (edges) incident out of a vertex v is the outdegree of v and it is denoted by $d^+(v)$. The number of arcs incident into v is its indegree and is denoted by $d^-(v)$. The degree of a vertex v is the sum of both: $d(v) = d^+(v) + d^-(v)$. For instance in Figure 29, the indegree of e_{t_2} is 2 and the outdegree is 1.

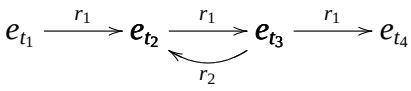


Figure 29: Cyclic graph example.

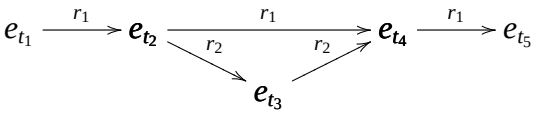


Figure 30: Acyclic graph example.

A query on data models seeks objects of interest. When it needs to merge multiple relations, the query should repeatedly navigate over relations and concatenate objects. Similarly a *walk* in a graph is an alternating sequence of vertices and edges, beginning (*origin*) and ending (*terminus*) with a vertex in which each vertex is incident to the two edges that precede and follow it in the sequence; the vertices that precede and follow an edge are the end vertices of that edge. The *length* of a *walk* is the number of edges that it goes through.

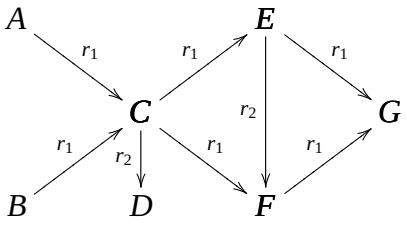


Figure 31: Graph walk example. For instance, the length of a walk through ($A \to C \to F \to G$) is 3.

7.2 Analysis on data models using graph theory

A relational database [30, 32, 33, 26] is an incarnation of first-order relations in which a node is a rectilinear table in rows or columns and an edge is a foreign key to refer to other node as shown in Figure 28. In this architecture, each table record encapsulated in one node has a strict relation to other nodes in a predefined way. The physical meaning of relations between nodes could be different in many cases. However relational models do not support any way to label such a relation.

In fact, relational databases focus on the size and speed of the data operation instead of the degree of freedom in modeling the application data model. Relational databases are therefore adequate for data that exhibit a great deal of regularity with a relatively low degree of complexity.

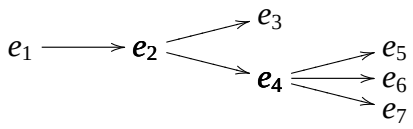


Figure 32: Tree-structured information relations.

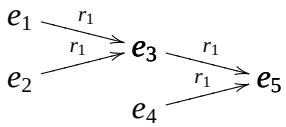


Figure 33: Many-to-one relation is not allowed in relational model.

In contrast to the rectilinear structure of relational databases, XML [13] is more like a tree structure, as shown in Figure 10, and suitable for modeling semistructured data objects such as HTML or general text documents. XML is an open standard that is self-describing and easily readable by both humans and machines. It is vendor-neutral and extensible. These features have led to its adoption in numerous industries. Most XML usage falls into one of three categories: (1) Data exchange and integration, (2) application development, and (3) content and its metadata management.

In addition to its representational power in information exchange, XML is also gaining acceptance as a data storage method. When XML is used as a storage system, it also needs a concept of schema to access and navigate XML documents because we cannot manually parse XML documents every time we access them. However, different data models should be stored into separated XML documents using different schemes because one XML document has only one root node. Such a collection of different XML documents becomes an information forest. Queries directed into such a forest need separate query commands for each tree object. They also need an additional standard called XInclude [99], which is not

yet fully supported by the leading Web browsers. Considering all of these situations, we cannot claim that XML reduces the complexity of information retrieval.

7.2.1 Relations

In a graphical representation, an edge between two nodes represents the connection, and in a physical data model, an edge represents the referential key between two data objects. In Chen's entity-relation (ER) model [27], an event is considered a connector between one-to-many relations. Thus, in a relational database, the role of an event within such a definition is a planar relation between fixed index keys as a constraint on reference. Their meaning is not labeled, and the connection between tables permits only a one-to-one connection.

```
<family>
  <name husband_of="B" father_of="C1" father_of="C2">A</name>
  <name wife_of="A" mother_of="C1" mother_of="C2">B</name>
  <name daughter_of="A" daughter_of="B" older_sister_of="C2">C1</name>
  <name daughter_of="A" daughter_of="B" younger_sister_of="C1">C2</name>
</family>
```

Figure 34: The XML standard does not support the above N-to-1 relations.

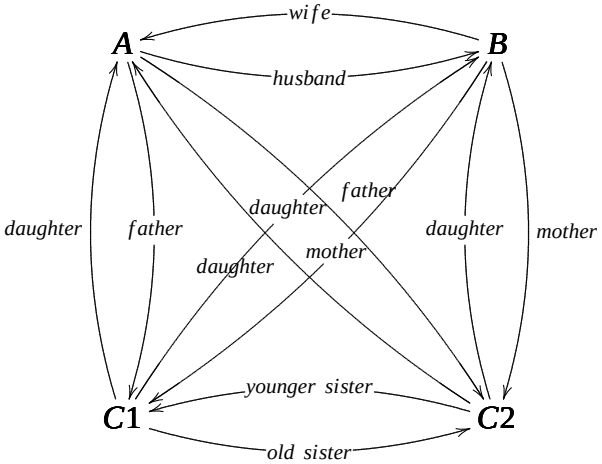


Figure 35: A complete graph for family relationships.

When it is used as a data model with a specified schema, XML also has similar limitations. Figure 34 exemplifies one case using XML in which repeated attributes for one

element are not supported. Thus, representation of the complete graph[1] as depicted in Figure 35 in one depth is not possible in XML. To simulate multiple attributes using XML, nested, and hierarchical modeling approaches are necessary, as exemplified in Figure 36. This example, however, increases tree depth and in turn, significantly slows query speed.

```
<family>
 <person>
   <name>A</name>
   <relations>
     <husband_of>B</husband_of>
     <father_of>C1</father_of>
     <father_of>C2</father_of>
   </relations>
 </person>
 <person>
   <name>B</name>
   <relations>
     <wife_of>B</wife_of>
     <mother_of>C1</mother_of>
     <mother_of>C2</mother_of>
   </relations>
 </person>
 <person>
   <name>C1</name>
   <relations>
     <daughter_of>A</daughter_of>
     <daughter_of>B</daughter_of>
     <older_sister_of>C2</older_sister_of>
   </relations>
 </person>
 <person>
   <name>C2</name>
   <relations>
     <daughter_of>A</daughter_of>
     <daughter_of>B</daughter_of>
     <younger_sister_of>C1</younger_sister_of>
   </relations>
 </person>
</family>
```

Figure 36: Nested hierarchical relations for XML.

As described above, the degree of freedom in data modeling is a database system design factor in balance with the regularity and performance of data models.

7.3 An e-group node revisited

Let us revisit the e-group node introduced in Section 3.4.2 in more detail in terms of implementation. In the E-model, an e-group node has a distinct identifier, and its e-sdtype has a

[1] A complete directed graph has an arc from every node to every other node. In this example, an arc to itself is ignored for simplicity.

complete directed graph has an arc from every node to every other node. In this example, an arc to itself is ignored for simplicity. specific name "*_eGroup*", and it raw data value is a global identifier like UUID in 36 characters [88] or a short UUID in 64 bits integers. In the E-model, an e-group node is a unit of query and algebraic operations (ex. walk, copy, move, split, and merge.) This is useful in logical operations and also for importing or exporting e-nodes from and to external data models. An e-group node is also a universal identifier that allows direct access to data objects without search constraints.

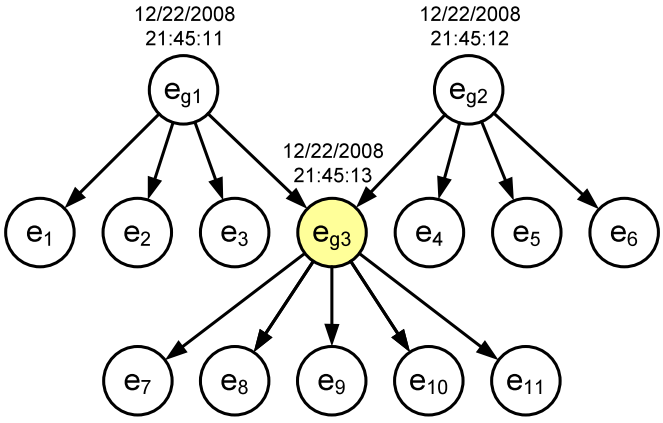


Figure 37: A new e-group node inherits an old e-group node.

The E-model strictly enforces the temporal relation between e-group nodes. The direction of flow of a relation between two e-group nodes should be from old to new e-group nodes. This reflects the natural behavior of a data archive in which facts on events are accumulated on top of existing facts. Figure 37 shows such an example in which e_{g_3}, which is registered later than e_{g_1} or e_{g_2}, is linked with them. For instance, in multimedia, assuming e_{g_1}as the result of audio processing and e_{g_2} as the result of video processing, then e_{g_3} would be a higher-level event detection similar to using voice identification and face recognition to identify a human.

7.4 Relational directed acyclic graph

A RDAG is a directed acyclic graph permitting multiple labeled relations between nodes. Figure 38 illustrates an example in which multiple relations (γ_1, γ_2) between two nodes also

support child nodes (e_{t_3}, e_{t_4}) that are offspring from the parent node e_{t_2}. We implemented this structure based on the E-model and named it RDAG.

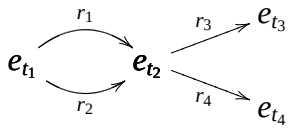


Figure 38: A relational directed acyclic graph.

Definition 7.1. A RDAG, D, is a directed acyclic graph. Assuming an incidence map, I_D is a set of trinary tuples, (u, r, v) that limits bounds the order of computational complexity, r represents the directed relation from the tail, u, to the head, v, of associated finite edges $r \in \mathcal{R}$. Multiple relations may exist in between u and v.

RDAG properties are listed below:

RDAG-1 RDAG nodes are e-group nodes e_g that are instances of e-categories. Relations e_r in RDAG are functional representation from the source (input) e-nodes to the target (output) e-nodes which are e-function nodes. The data type of both e_g and e_r is an identical e-node type.

RDAG-2 The acyclicity condition is achieved by limiting the flow of edges from old to new e-group nodes $\{ \mathit{timestamp}(u) \leq \mathit{timestamp}(v) \}$ and prohibiting self-relation that links one e-group node to the same e-group node. The age of an e-node is identified by its transactional timestamp that satisfies the rule. Self-referencing that creates an infinite loop is not allowed by the e-category definition in Definition 5.3.

RDAG-3 A RDAG is not a simple graph that permits only one edge between endvertices. The multiplicity N of an edge is the number of multiple edges sharing the same endvertices. The multiplicity of a graph, $N(R)$, is the maximum multiplicity of its edges. In a RDAG, $N(R)$ is assumed to be bounded: $N(R) \ll \infty$.

RDAG-4 The relation types depend on the application domain. They are assumed to be finite and known a priori. Therefore, the multiplicity of relations in a RDAG is bounded.

$$N(R) = M < \infty, \tag{33}$$

where M is a positive constant and is finite.

RDAG-5 A RDAG can be interpreted by its acyclic and labeled graph structure without schema definitions and thus, can fully support both unstructured information retrieval and structured queries based on associated e-categories.

Figure 39 shows an example of a RDAG in which multiple e-group nodes are connected in various relations. They are ordered in time because their node ID numbers from one to eight show their temporal order.

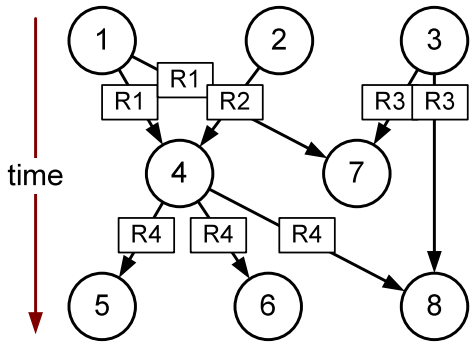


Figure 39: A relational directed acyclic graph sample.

7.5 A super e-node

Because a RDAG is acyclic, it may have a number of root nodes from which all offspring nodes originate. In the E-model, we named the upper-most root node of all e-group nodes super e-node, e_{se}. The super e-node is also an e-group node as an instance of one independent E-modelsystem. e_{se} is the first e-node that the E-model system creates as the earliest of all e-nodes. When a new instance of an e-category is registered, one unique e-group node, which is a parent (hierarchical parent) of all child e-nodes, is created and connected to its

parent e-group node. By proceeding in this way, all e-group nodes in a graph constitute a snowflake schema form [92] in which all nodes stem from the root node. The super e-node serves the following important purposes:

SE-1 It is the identity of one E-model system in the distributed network.

SE-2 The access identity of an e-node in the E-model system can be the composition of the super e-node, e_{se}, and the e-node ID, e_c. For instance, $[e_{se}, e_c]$ means that an e-node e_c is stored at the e_{se} system.

7.6 RDAG storage architecture

A graph is an abstract representation, and the approaches to implementing a graph as a storage system differ significantly from other approaches. In the E-model, a vertex becomes an e-node with relations to other e-nodes. In designing a storage system for graph data objects, if the length of a walk (i.e., the length of an information search path) is increased, then query speed will be decreased accordingly. This section studies an efficient way to store and retrieve a RDAG.

Well-known storage models for DAGs are (1) a materialized path or called a path enumeration model [119], (2) a nested set model of hierarchies [25] and (3) an adjacency list for graphs [52]. Let us review the features of each model using the sample graph depicted in Figure 39.

In the materialized path [119], each record stores the whole path to the root into Table 3, which has at least two columns of ID records and their paths. The table uses the sibling numerators instead of the node primary keys. However, Figure 39 is a forest composed of three trees because the materialized paths cannot simply represent the forest. Thus, Table 3 includes three tables, one fore each tree. Besides, a path in string representation raises many issues of computational inefficiency and object matching. If the paths are split into separate tables Table 3, then a separate query must be run for each table. This significantly decreases query performance.

Table 3: The materialized path expression of Figure 39.

EID	Path
E1	1
E4	1.1
E7	1.2
E5	1.1.1
E6	1.1.2
E8	1.1.3

EID	Path
E2	1
E4	1.1
E5	1.1.1
E6	1.1.2
E8	1.1.3

EID	Path
E3	1
E7	1.1
E8	1.2

Table 4 lists instances represented in a nested expression of Figure 39. The nested set model [25] is composed of intervals of integers, and it is well-defined to fit into a relational database and easily represents the hierarchical and categorized relations between instances. But it has problems similar to the materialized paths in forest data models because we need three tables. Also, the computational burden imposed in managing the nested set and its ongoing inefficiency makes the nested set model impractical to use for the frequent insertion of trees because for every insertion and update of the set, all intervals within the same set must be recomputed.

Table 4: The nested set expression of Figure 39.

EID	Left	Right
E1	1	12
E4	2	9
E7	10	11
E5	3	4
E6	5	6
E8	7	8

EID	Left	Right
E2	1	10
E4	2	9
E5	3	4
E6	5	6
E8	7	8

EID	Left	Right
E3	1	6
E7	2	3
E8	4	5

In addition to their computational efficiency, both the materialized path and nested set models in the above examples have two other limitations: (1) They do not support relational labels between instances, and (2) inheritance is limited to propagation from within the same category, which is not a feasible model in the handling of heterogeneous sources in complex processing.

Because of these limitations of the alternatives, our approach chooses the adjacency list described in the following section as the E-model storage architecture. An adjacency list is

the representation as a list of all the edges or arcs in a graph. Table 5 exemplifies Figure 39 in which adjacent nodes are listed in a row.

Table 5: The adjacency list of Figure 39.

EID	Path	EID	Path	EID	Path
E1	E4	E3	E7	E4	E8
E1	E7	E4	E5		
E2	E4	E4	E6		

7.6.1 RDAG storage model

This section further extends the above example to represent the relation between nodes. Let us start with a 2-tuple storage model depicted in Figure 40 and then compare it with Figure 41, which has a 3-tuple storage structure.

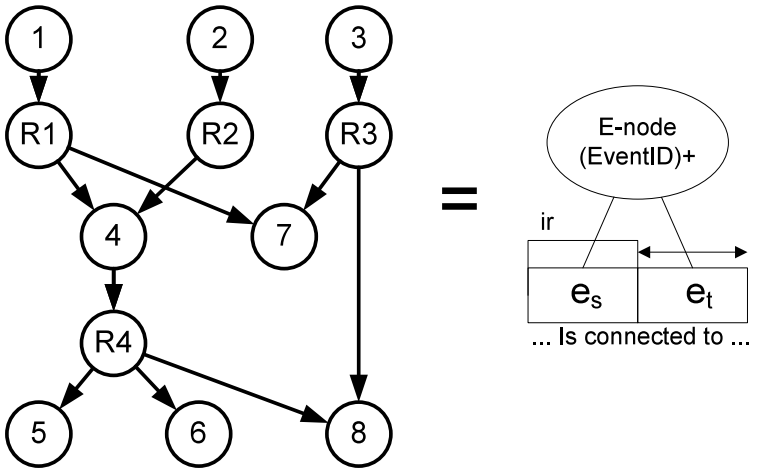


Figure 40: RDAG 2-tuple storage model.

The count of nodes between $\{e_s, e_t\}$ is labeled as N, while the length of the 2-tuple adjacency list is represented as L_2, and L_3 for the 3-tuple adjacency list model. In a 2-tuple storage model, the path is a transitive link of nodes. If a storage model needs to support multiple types of relations, a relation object should be a distinct linking of two nodes in a 2-tuple adjacency list. For a 3-tuple adjacency list, a relation object is located between two nodes, and the path length will not increase. As a summary, the path length can be

back-calculated as shown below.

$$\begin{aligned} L_2(e_s, e_t) &= 2N - 1, \\ L_3(e_s, e_t) &= N. \end{aligned} \tag{34}$$

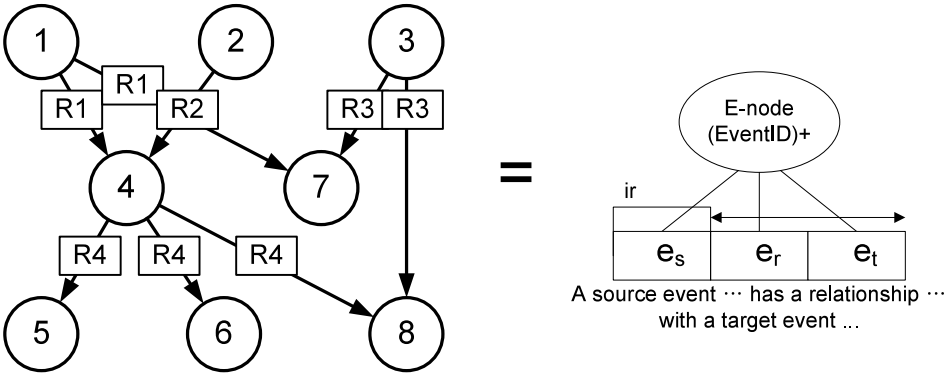


Figure 41: RDAG 3-tuple storage model.

A 3-tuple adjacency list has an additional tuple, whereas a 2-tuple storage model increases the path length significantly. The 2-tuple adjacency-list model needs two self -joins to query two nodes with a specified relation. This significantly slows the query speed and increases query complexity. Consequently, based on the following observations, we chose a 3-tuple adjacency list as the data structure:

AL-1 A relation e-node is a 2-clique[2] by which 3-tuple adjacency lists can represent the full relation between two e-nodes.

AL-2 This also ensures that relation e-nodes are always placed at the middle entity of an adjacency list that facilitates the grouping of e-nodes in a special relation.

AL-3 If a relation e-node is not considered, then it is equivalent to the flat-relational adjacency model that assures the compatibility with existing data models.

Higher tuple models will not be considered in this work because the first-order relation composed of source, target, and their relation object serves as complete representation of

[2] A clique in a graph is a set of pairwise adjacent vertices

the higher-order relation in complex data models. In a RDAG, multiple paths may exist between two e-nodes. Thus, if a path query includes specifications on relations, the number of edges between two e-nodes formed by the use of such expressions is always equal to or less than the total edges:

$$|\mathcal{P}(e_s, R, e_t)| \quad \leq \quad |\mathcal{P}(e_s, e_t)|, \tag{35}$$

where R is the set of ordered relations, $\mathcal{P}()$ is a path operator finding all paths from the source node e_s to the end node e_t, and || is the size of a graph counting edges in the path.

7.7 RDAG query

A query on a RDAG is formulated in two stages: (1) locating source nodes of interest and (2) walking though the graph to find target nodes. This section reviews special RDAG features for a query. In a RDAG, the query function is a transitive first-order relation query. Search constraints can be applied for each relation query. Because all objects and definable constraints are described in detail in Chapter 3, this section focuses on the constraints for graph walks.

7.7.1 RDAG path walk

A walk in a graph is an alternating sequence of nodes and edges. The path length then consists of the count of visited edges. As explained before, e-group nodes are distinct, whereas e-nodes that are child e-nodes of an e-group node can be shared by e-group nodes. Figure 42 (b) illustrates this example in which e-group nodes e_4 and e_6 are connected through e_5.

In finding such connections, the limit of the path length is based on the super e-node. This is because in a RDAG all top-level e-group nodes are connected to the super e-node. This does not include the semantic connection, because in Figure 42 the relation R_1 between the super e-node and top e-group nodes differs from that of relation R_4 between e-group nodes and their child e-nodes. From these facts of the RDAG acyclic property, the search

length between two e-nodes has a upper boundary by Lemma 7.1.. The path search ends when it reaches a leaf e-node.

Lemma 7.1. If two e-nodes are connected by a shared e-node, the path length between two e-nodes is less than the sum of the path lengths from each e-node to the super e-node:

$$|P(e_1, e_2)| \quad \leq \quad |P(e_s, e_1)| + |P(e_s, e_2)|, \tag{36}$$

where e_s is the super e-node,, e_1 and e_2 are each source and target e-group nodes for finding a path, and R is a set of ordered relations. $P()$ is a path operator in finding the shortest path from e_1 to e_2. || calculates the size of a graph by counting edges in a set.

Proof. Finding a path between two e-nodes of different e-categories, requires locating the common e-node that connects both e-nodes via an e-group node. In this case, the maximum path length of each e-node is bounded less than the length from the e-node to the super e-node, $|P(e_s, e)|$. Therefore, the sum of both lengths is less than or equal to like Eq. 36. □

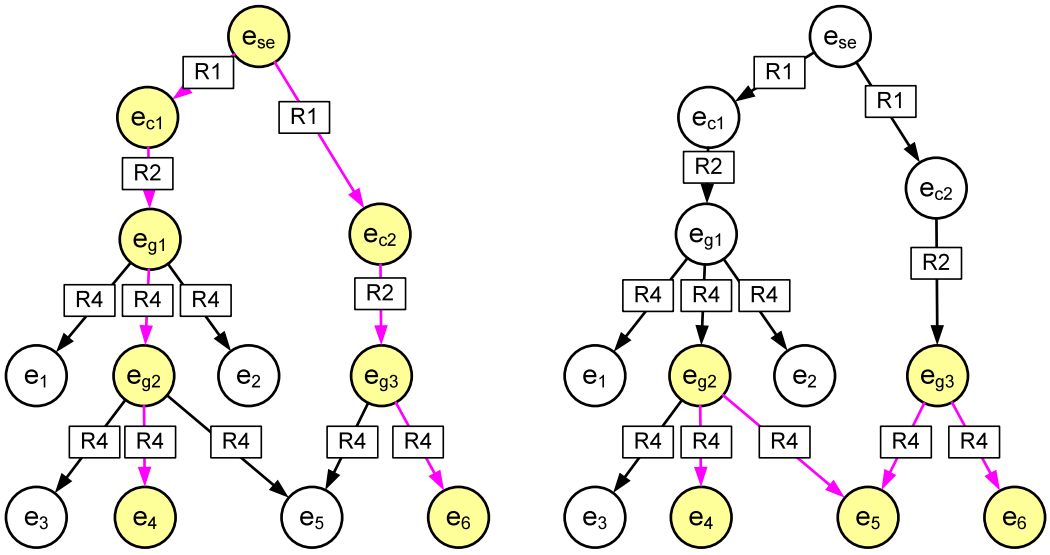


Figure 42: Path query example between two e-nodes in RDAG.

Figure 42 shows one example of finding a path between two e-nodes e_4 and e_6 through a RDAG. The path in Figure 42 (a) walks through the super e-node e_{se} and its walk history

is $\{e_4 \rightarrow e_{g2} \rightarrow e_{g_1} \rightarrow e_{se} \rightarrow e_{g3} \rightarrow e_6\}$ in length 5. For a path in Figure 42 (b), it finds the route through a shared e-node that walks through $\{e_4 \rightarrow e_{g2} \rightarrow e_5 \rightarrow e_{g3} \rightarrow e_6\}$ in length 4. This satisfies Lemma 7.1 by which path (b) length 4 is less than the sum of length 5 of each e-node to the super e-node. This example also shows that Lemma 7.1 is useful in precalculation for optimization (ex. size and number of stacks) before a walk. Path (b) in Figure 42 is the shortest path, which does not include the super e-node, from e_4 to e_6 that walks through the shared e-node e_5. Hence, by Lemma. 7.1, the existence of a shared e-node is the condition for finding the shortest path between e-nodes.

In fact, finding a path between related e-nodes is not limited to finding the first-order relation between e-nodes. Figure 43 and Figure 44 show two other examples in which an example (a) explains relations though a high-order parent e-group node and (b) explains relations through high-order sibling e-nodes. The order of relations is the distance from the source e-node to the target e-node. Physical interpretation of these relations could differ by domain applications. For instance, using Figure 43, two e-nodes e_4 and e_{10} areconnected by the grandparent e-group node e_{g_1}, and its length is shorter than the path through the super e-node. In object-oriented interpretation, their connection can be understood as the offspring child objects originated by the same parent object. Thus a similar theory can be formulated as below.

Theorem 7.1. The shortest path exists between two e-nodes if both e-nodes share the common e-group node.

Proof. The proof is straightforward because all e-group nodes are offspring from the super e-node; as a result, as demonstrated below, the path length between them should be always shorter than the path through the super e-node:

$$|P(e_1, R_4, e_2)| \quad = \quad |P(e_s, R_1, e_1)| + |P(e_s, \{R_1|R_2\}, e_2)| - 2|P(e_s, \{R_1|R_2\}, e_p)|, \qquad (37)$$

where a set of the relations included in the path are specified as a set in Boolean expression.

□

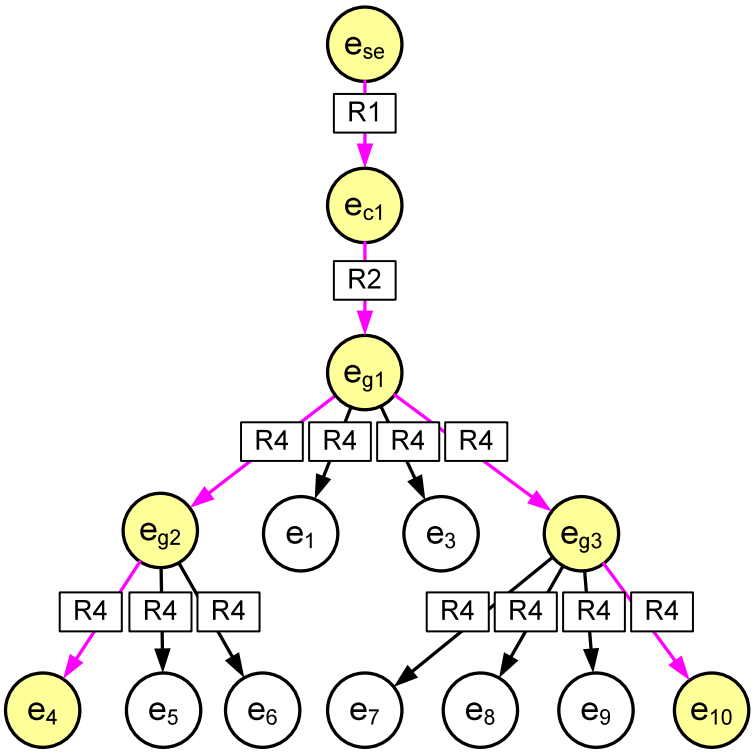


Figure 43: Path through high-order relations.

Figure 44 shows the other case in which two e-nodes are related by a shared e-node through sibling e-nodes e5 and e7. A common e-group node, e_{g_3}, links them as its child e-nodes. In this case, a common e-group node does not exist from the viewpoint of physical interpretation. No structured relations exist between two e-categories that are each the associated schema object of an e-group node. This can be understood as one of the following cases:

SS-1 Concatenation of first-order common e-group relations for higher-order relations.

SS-2 Newly linked relations revealed from real-world instances.

SS-3 Misusage of an e-node name that does not distinguish its physical identity and relations with other e-nodes.

Remember that an e-node in the E-model is an instance of an e-sdtype, which has a fixed symbol and a raw data type. Therefore, if two e-group instances share the same e-node,

their e-categories have the same e-sdtype as their child entity. This is similar to the concept of a foreign key in a relational model [33] and to IDREF in XML [15] , except the constraint that that the targeted data object relies on the source data object for its validity. Such constraints will be applied in a case such as Figure 43 in which all relations and their hierarchical instances depend on predefined parent-child e-categories.

Theorem 7.2. A path exists between two e-nodes if the parent e-group nodes of each e-node share more than one common e-node.

Proof. If the search path is limited to a RDAG, then the condition for finding the path between two e-nodes lies in finding a common e-group node from which both e-nodes originate. In this case, the maximum path length between e-nodes is bounded less than the length from the e-node to the super e-node, $|E(e_s, e)|$. Hence, the sum of the length of both paths is less than or equal to that in Eq. 36. □

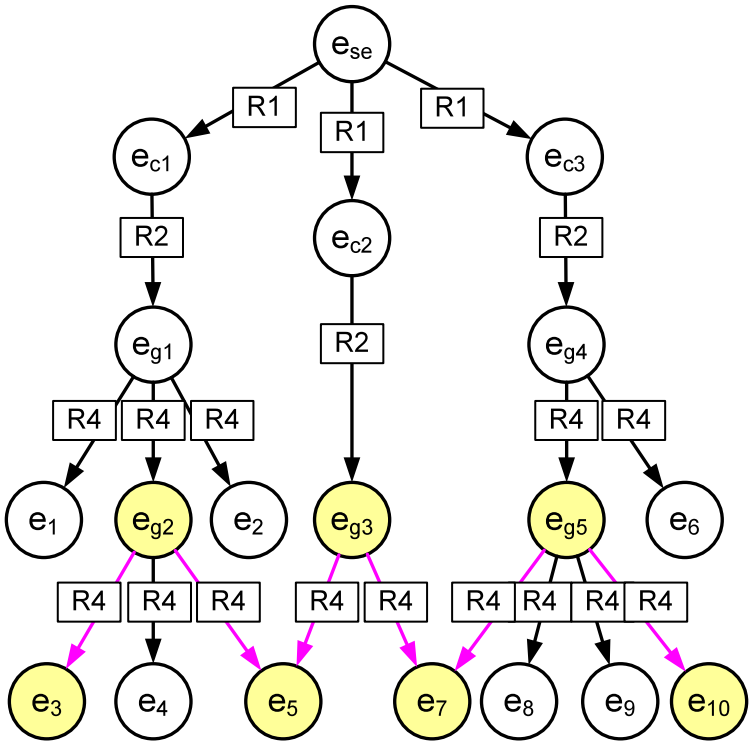


Figure 44: Path through connected sibling e-nodes (ex. Find all events at a specific valid time).

7.7.2 Find path through a shared e-node: M-algorithm

Based on the mathematical analysis of a RDAG graph search, let us formulate an al-gorithm for finding such related e-nodes as a way to find a path between two e-nodes through a shared e-node. Figure 45 depicts the flow of queries from A to E for finding relations between two e-nodes. This figure shows a case in which two e-nodes (A and E) do not share the same group node, but they are related by another e-node (C). This query is independent of the associated categories since it looks for the connected and shared e-node independent to its associated e-category. In other word, this query is an unstructured query.

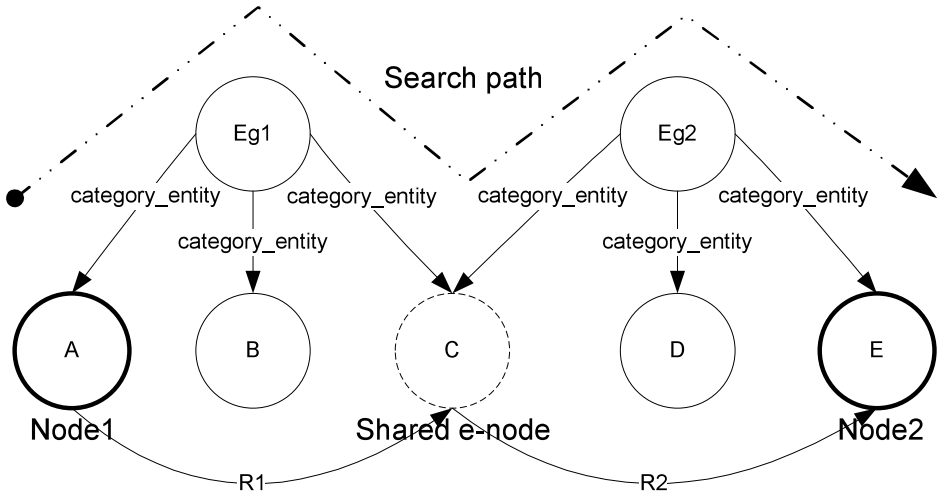


Figure 45: Related information search over group nodes.

From given input data values and types, we first find the seed e-nodes, E_{p_1} and E_{p_2} to search for a path between the two. Then we proceed to find other e-nodes under the same category. If no matches are found in the sibling e-nodes, the search moves up one level to the parent e-group node. Through this bottom-up manner, the search keeps moving upward until it arrives at the super e-node. This algorithm is designated the M-algorithm. Its set

representation is formulated at Eq. 38.

$$
\begin{aligned}
E_{p_1} = C(f_e(A)),\ E_{p_2} = C(f_e(E)),\\
E_c^{p_1} = S(E_{p_1}),\ E_c^{p_2} = S(E_{p_2}),\\
f_e(c) = E_c = E_c^{p_1} \cap E_c^{p_2},
\end{aligned}
\tag{38}
$$

where f_e is a function to find e-nodes of a given data value (A and E in Figure 45). C is a function to retrieve a category e-node. S is a function to find all entity e-nodes from a given e-group node, and E_c is a result set of e-nodes that $E_c^{p_1}$ and $E_c^{p_2}$ commonly share.

Because a RDAG permits multiple relations between e-nodes, the same e-nodes may be connected in multiple relations. Such multiple relations can be limited to specific ones to reduce the computational load. In this query, even without any knowledge of an e-node symbol or value, a user can query a set of source and target e-nodes using a specific relation. This is a simple and powerful solution when a user is looking for e-nodes in special relations so as to retrieve all related e-nodes.

In addition, all data objects in a RDAG are e-nodes, and they are the composition of c-data objects. The keyword query method is thus equivalent to all RDAG objects without prior knowledge of their relations. This means a keyword query for all e-categories is very efficient, and once found, the navigation through the connection in a RDAG is straightforward without looking for additional metadata. In other words, e-group nodes of a RDAG are instances of e-category instances, and the M-algorithm is an unstructured query method for structured objects.

To search for more than connected data objects, temporal relations and their operators [5] can be naturally applied to queries. Because all e-nodes have a transaction timestamp, temporal relations can be used in searching for e-nodes on the timeline. If a user wants to search for information using a valid time [84], then the query can be extended to find the timestamp-type e-nodes under the same e-group node. This permits a query to be expanded over other e-nodes that have timestamp-type c-data with the same symbol c-data

within specified temporal ranges. In addition, semantic expansion using a shared dictionary (See Section 8.5) is feasible because it is limited to finding e-nodes to search at the first step.

Algorithm 5: M-algorithm - Homogenous (RDAG)

Data: Two data and/or name values to find the path between them, I_x and I_y.
Data: Constraints on each value, L_x and L_y.
Result: The path set, P_{xy}, stores all possible routes from $\{e_{p_x} \in E_{p_x}\}$ to $\{e_{p_y} \in E_{p_y}\}$.
begin
REM Find the seed e-nodes from a given input data value.
$E_{p_x} \longleftarrow FindNodes(e_d(I_x), L_x)$
$E_{p_y} \longleftarrow FindNodes(e_d(I_y), L_y)$
REM Retrieve the parent e-group nodes.
$E_c^{p_x} \longleftarrow GetParentNodes(E_{p_x})$
$E_c^{p_y} \longleftarrow GetParentNodes(E_{p_y})$
while $|E_c^{p_x}| \neq \emptyset$ *and* $|E_c^{p_y}| \neq \emptyset$ **do**
REM Check the connection between two e-nodes, e_{p_x}, e_{p_y}.
for $e_{p_x} \in E_{p_x}$ **do**
for $e_{p_y} \in E_{p_y}$ **do**
if $e_{p_x} = e_{p_y}$ **then**
REM Store path.
$P_{xy} \Leftarrow StorePath(e_{p_x}, e_{p_y})$
else
$E_c^{c_x} \longleftarrow GetChildNodes(e_{p_x})$
$E_c^{c_y} \longleftarrow GetChildNodes(e_{p_y})$
REM Find the common e-node that two e-group nodes share.
$e_{c_x} = E_c^{c_x} \bigcap E_c^{c_y}$
if $IsNotNull(e_{c_x})$ **then**
$P_{xy} \Leftarrow StorePath(e_{p_x}, e_{c_x}, e_{p_y})$
REM When it reaches the super e-node, then stop.
$E_c^{p_x} \Leftarrow GetParentNodesExceptSuperNode(e_{p_x})$
$E_c^{p_y} \Leftarrow GetParentNodesExceptSuperNode(e_{p_y})$
end

The above ideas are formulated as the M-algorithm. The M-algorithm is a method to find the path between two e-nodes through a shared e-node. We developed the M-algorithm in two versions. The Algorithm 5 version is for a case that follows a RDAG within associated e-categories and under one super e-node. The Algorithm 6 version allows the search to walk over heterogeneous categories in which sibling e-nodes are associated.

Algorithm 6 also permits a search across different super e-nodes, in other words, a search over the distributed network.

Several functions are defined for the M-algorithm. *FindNodes* finds e-nodes based on given reference values of a symbol and/or a data value. Because one e-node is composed of two c-data objects and one transaction timestamp, search constraints can be specified on these three entities. *FindNodes* is based on a Search-by-Value (SBV) algorithm that according to our E-model system, has superior performance compared with popular storage models (See Section 11.4). *GetParentNodes* retrieves e-group nodes from a given e-node based on the RDAG network. It returns the first parent e-node and multiple e-nodes because one e-node may be referenced multiple times by different e-group nodes. *GetChildNodes* returns all child e-nodes from a given parent e-group node.

In the case of the heterogeneous M-algorithm, the search pool has no prenavigation boundaries. Also, when the search moves over different super e-nodes, the temporal constraint in one RDAG does not apply to another RDAG. Hence, in the algorithm, several constraints are applied to searching e-nodes. For instance, T_r limits the time range. When a user want to find "near" events or previous events only, then T_r can work as the temporal window to limit the e-nodes in the search. If a user want to search over special relations like specific spatio-temporal relations, then such conditions can be modeled as L_r. Both algorithms finish searching when their trace moves up to an associated super e-node.

Algorithm 6: M-algorithm - Heterogenous (RDAG)

Data: Two data and/or name values to find the path between them, I_x and I_y.
Data: Constraints on values, L_x and L_y.
Data: Constraints on relational time ranges, T_r.
Data: Constraints on relations, L_r.
Result: The path set, P_{xy}, stored all possible routes from $\{e_{p_x} \in E_{p_x}\}$ to $\{e_{p_y} \in E_{p_y}\}$.
begin

REM Find the seed e-nodes from a given input data value.
$E_{p_x} \longleftarrow FindNodes(I_x, L_x)$
$E_{p_y} \longleftarrow FindNodes(I_y, L_y)$
REM Retrieve the parent e-group nodes.
$E_c^{p_x} \longleftarrow GetParentNodes(E_{p_x})$
$E_c^{p_y} \longleftarrow GetParentNodes(E_{p_y})$
1 **while** $|E_c^{p_x}| \neq \emptyset$ *and* $|E_c^{p_y}| \neq \emptyset$ **do**
REM Check the connection between two e-nodes, e_{p_x}, e_{p_y}.
2 **for** $e_{p_x} \in E_{p_x}$ **do**
3 **for** $e_{p_y} \in E_{p_y}$ **do**
if $e_{p_x} = e_{p_y}$ **then**
REM Store path.
$P_{xy} \Leftarrow StorePath(e_{p_x}, e_{p_y})$
else
$E_c^{c_x} \longleftarrow GetChildNodes(e_{p_x}, t \in T_r)$
$E_c^{c_y} \longleftarrow GetChildNodes(e_{p_y}, t \in T_r)$
REM Find the common e-node that two e-group nodes share.
$e_{c_x} = E_c^{c_x} \bigcap E_c^{c_y}$
if $CheckConstraints(e_{c_x}, L_r)$ **then**
$P_{xy} \Leftarrow StorePath(e_{p_x}, e_{c_x}, e_{p_y})$
REM Expand the search over sibling e-nodes' parent e-nodes.
REM When it reaches the super e-node, then stop.
$E_c^{p_x} \Leftarrow GetParentNodesExceptSuperNode(E_c^{c_x})$
$E_c^{p_y} \Leftarrow GetParentNodesExceptSuperNode(E_c^{c_y})$

end

CHAPTER VIII

IMPLEMENTATION

Implementation of a new data model requires consideration of many things beyond just theoretical formulations. Many graph data models have stopped at the theoretical proposal stage without being implemented or published [7]. Let us first check two critical conditions for data model implementation.

The first condition is whether a new data model can be built atop an existing database system. A database system needs many basic features like (1) a fundamental storage model, (2) a query language for data operation, (3) network features for server-client communication, and (4) transactional stability for enterprise purposes. If a new data model requires building a new database engine from scratch to provide all of these features, the time, effort, and cost of development will be significant and validation of its practical applicability will take a long time.

The second condition is whether such a new data model is interoperable with existing data models. Interoperation with existing data models is essential because many data objects are stored in disparate data models with various types of storage systems. If all data objects need to be converted for a new data model, its application domains will be quite limited. Hence, a new data model should support (1) import or export of other data models, (2) provision of views on the current status formulated for other data models,or (3) support open database for database connectivity.

The actual implementation of the E-model for the data storage can be accomplished in various ways. In our implementation, in light of the above design criteria, the E-model prototype is developed on top of the existing relational database to assure interoperability with existing data models and also to save time and costs that will reduce the time-to-market.

All the functions of this E-model are implemented as SQL stored procedures and SQL functions. This reliance on SQL ensures the suitability of the E-model for most relational databases. This chapter describes the details of our implementation.

8.1 C-data objects

An abstract c-data storage model was introduced in Section 3.5. Its implementation as a storage model is depicted in Figure 46. The c-data object in this design is a composite index encapsulating disparate raw data types. However, the c-data design is not limited to Figure 46, because of the following design criteria:

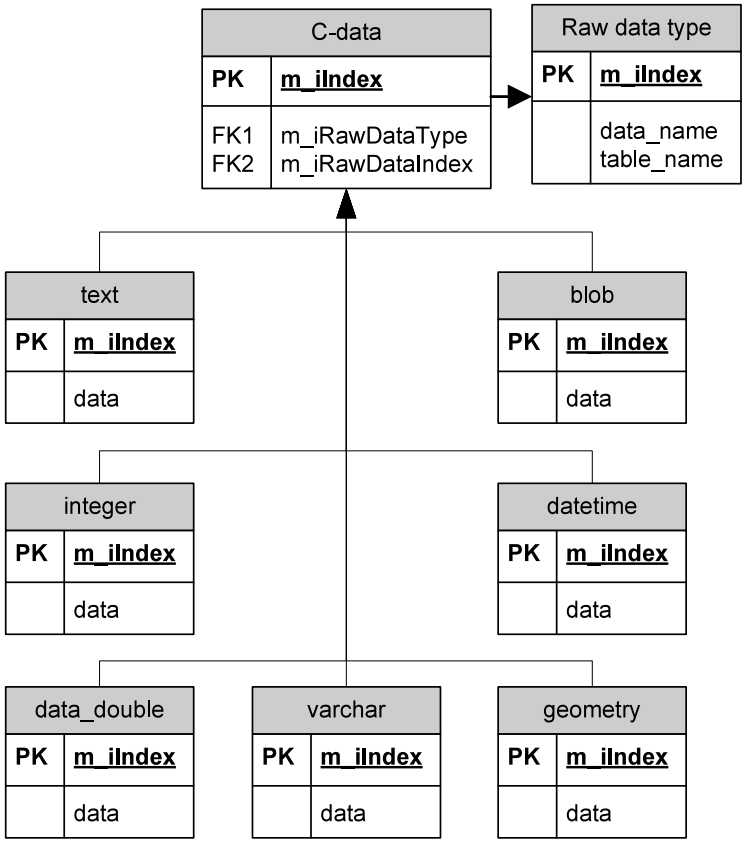


Figure 46: C-data storage architecture.

CD-1 The number of raw data types affects query speed. For instance, performance of a SBV query that lacks raw data type specifications requires that this query examine

all raw data tables. Consequently, in terms of query performance, fewer raw data type queries are preferred.

CD-2 Instances of one raw data type do not need to be stored into only one table. Based on the application demands such as security, speed, safety, or distributed storage, separate tables can be attached for the same raw data type..

CD-3 Raw data types needs not be limited to what the database system supports. Current c-data design assigns one table to each raw data type. This means that any table can be the raw data type of c-data if the operations in the Table 6 object can be supported for the raw data table. Thus, any ordered list or a composite set table in which one row is composed of multiple columns can be the raw data type.

Table 6: The list of c-data APIs (* is optional).

Direction	Name	Arguments	Return
Store	PutCData	Value, Raw data type	CData ID
Store	PutRawData	Value, Raw data type	Raw data ID
Retrieve	GetCData	Value, Raw data type*, Constraint*	CData ID
Retrieve	GetRawData	Value, Raw data type, Constraint*	Raw data ID

Because we built the E-model system atop a relational database, all APIs can be internally translated into SQL statements. This means that complex conditions that check the validity of data in a search can be supported through direct SQL statements. The APIs in the table are simplified interfaces for external access for applications that cannot directly connect to the database server.

8.2 E-node objects

An e-node is a 3-tuple object in which a name and a value of an e-node refer to c-data objects. Its transaction time is logged at the timestamp of an e-node. The E-model relation object is composed of three e-nodes and the necessary APIs are listed in Table 7. Search constraints for each API are introduced in Section 4.2 and Section 4.3.

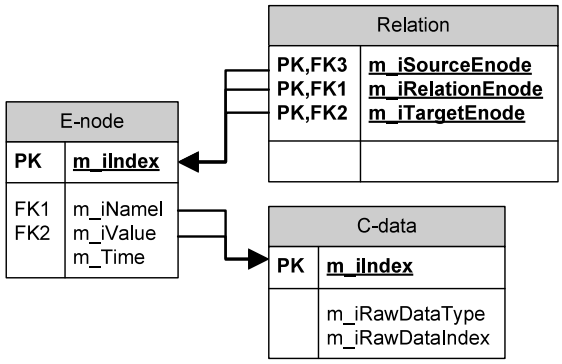


Figure 47: E-node storage architecture.

Table 7: The list of e-node APIs (* is a conditional constraint).

Direction	Name	Arguments	Return
Store	PutENode	Name c-data, Value c-data	E-node IDs
Store	PutRelation	Source, Relation, Target e-nodes	
Retrieve	GetENode	Name c-data*, Value c-data*	E-node IDs
Retrieve	GetRelatedENode	Source*, Relation*, Target* e-nodes	E-node IDs

In the E-model schema, objects and their instances are treated equally as e-nodes. To differentiate them, E-model schema elements use reserved names and predefined raw data formats as listed in Table 8.

Table 8: Special e-node types.

Type	Name	Value format
e-group	_eGroup	Short UUID
e-category	_eCategory	Category ID
e-function	_eFunction	Function name
e-sdtype	_eSDType	SDType ID

As mentioned earlier, relations between e-nodes in the E-model constitute RDAG and e-category instances. To support such a structure, e-functions are reserved for RDAG and data model interoperation. The e-functions of the current E-model system prototype are listed in Table 9.

Table 9: Reserved relation e-nodes (All have *_eFunction* as its name).

Value	Comment
_eSuper_Relation	Add a child element of a super e-node
_eCategory_Entity	Add entities to an e-category
_eFunction_Entity	Specify child entities of an e-function
_eFunction_Input_Entity	Add input entities of an e-function
_eFunction_Output_Entity	Add output entities of an e-function
_eSDType_Entity	Specify entities of an e-sdtype
_eGroup_Node	Add an instance of an e-category
_eRelation_Row	Add a child relational row element to an e-group
_eRelation_Column	Add a child relational column element to an e-group
_eRelation_Foreignkey	Equivalence relation from the source to the target e-nodes
_eXML_Parentchild	Add a child XML element to an e-group
_eXML_Sibling	Add a sibling XML element to an e-group
_eXML_Property	Describe an XML element property

8.3 Integrity constraints

In databases, ACID [54] is a set of properties (atomicity, consistency, isolation and durability) that guarantees that database transactions can be reliably processed. The E-model supports ACID in two aspects:

IR-1 Our work encapsulates application-dependent factors using the c-data and e-node concept. We will not be concerned with raw level data ACID since integrity rules for c-data objects are supported by the back-end storage system.

IR-2 Newly updated, deleted, or inserted data schema objects in typical storage systems prohibit existing data interpretation and thus invalidate data consistency. In the E-model, such schema morphism or versioning problems are relieved by inherent e-node tem-poral properties that constitute an e-category with complete transaction time records of updates. As for schema morphism protection, changes in the e-category should not delete old elements, and updated child objects will be registered as new children. Thus, in retrieving associated data, timestamps of e-categories and temporal timestamps of e-group nodes work as references to retrieve an ACID-conformed category model and its associated data objects.

8.4 RDAG address allocation rules and storage types

E-group nodes, which are instances of an associated e-category, constitute a RDAG, and they are linked in a temporal order from old to new e-group nodes. Hence, the temporal siblings are located next to each other if they are direct descendents of a parent e-group node. This fact can be used to speed the graph walk process, considering that an adjacency linked list, which is the storage architecture of a RDAG, causes sequential *SELECT* queries to walk through the path. Thus, when searching child e-group nodes from a given parent e-node, we can use the property of a RDAG from a given source e-node, e_s, because the index of e-nodes to visit is limited by their temporal order:

$$ID(e_s) < ID(e_g), \tag{39}$$

where $t(e_g) \leq t(e_s)$. And if it is a direct child e-node, then its index should be bigger than:

$$ID(e_c) \geq ID(e_s) + 1, \tag{40}$$

where 1 is the unit of an e-node address by which a new e-node index increases. If Eq. 40 satisfies an equal condition, then the query becomes deterministic and does not need a query on the e-node relation table. This is because the next e-node to visit, ec , can be directly retrieved from the e-node ID: $ID(e_c) = ID(e_s) + 1$. This enhances the RDAG query speed significantly because this does not need a repeated *SELECT* to search child e-nodes.

To satisfy the above conditions, the address space of e-group nodes should be independent of that of other e-nodes, i.e., e-nodes that are the nodes of a RDAG should have separated address space to avoid interference with the IDs of pure e-nodes. Also, within the e-group address space, each category and its instance e-group nodes should have separately allocated space. Implementation of this algorithm depends on the back-end storage system and will be added in the future to the current E-model prototype system.

8.5 Semantic object annotation for information retrieval

In complex data models, the semantic meaning of an object is important in indexing and accessing data objects. However, for complex objects, one symbol (or one name in text form) for a data value is not enough to convey the full semantic meaning of a data object. But linking more data symbols to the data object is heuristic and also adds complexity to the system in modeling their structures and handling their properties. Hence, in practice a common dictionary is built to extend the semantic meanings of data objects. When such a common dictionary has structured rich relations between data objects that can be shared within communities, it is referred to as an ontology (Interested readers are referred to [55, 127, 51]).

In fact, the SemanticWeb organization very recently has been working on the RDF/OWL representation of WordNet [133]. This approach comes from the need in ontology management [104] for popular semantic database approaches [69, 110, 127] that use ontologies, domain-specific name space and dictionaries. As ontologies are getting widespread and are collaborative, inconsistency in the name of an element in an ontology definition can be semantically confusing across different groups of users. Thus, a common dictionary has become a necessity for ontology management And they adopted WordNet to clarify the semantic synset of words.

Ontology is mostly represented using XML (See OWL [127, 100] and DAML [109, 66]), which the E-model can naturally export to and import from, which is a topic discussed in detail in Section 9.2. This section is, however, more focused on the topic of a common dictionary with the aim of appending semantic extensions on linked objects (in the ontological concept). Let us assume that a symbol is the name of an object and should be defined distinctively within the community of users. The problem then becomes how can users realize exactly which texts to search. In keyword-type queries, we would consider various derived words over related synsets to find words of interest. For instance, if a user wants to query the keyword, *marry*, then related words may be like *married* or *marriage*.

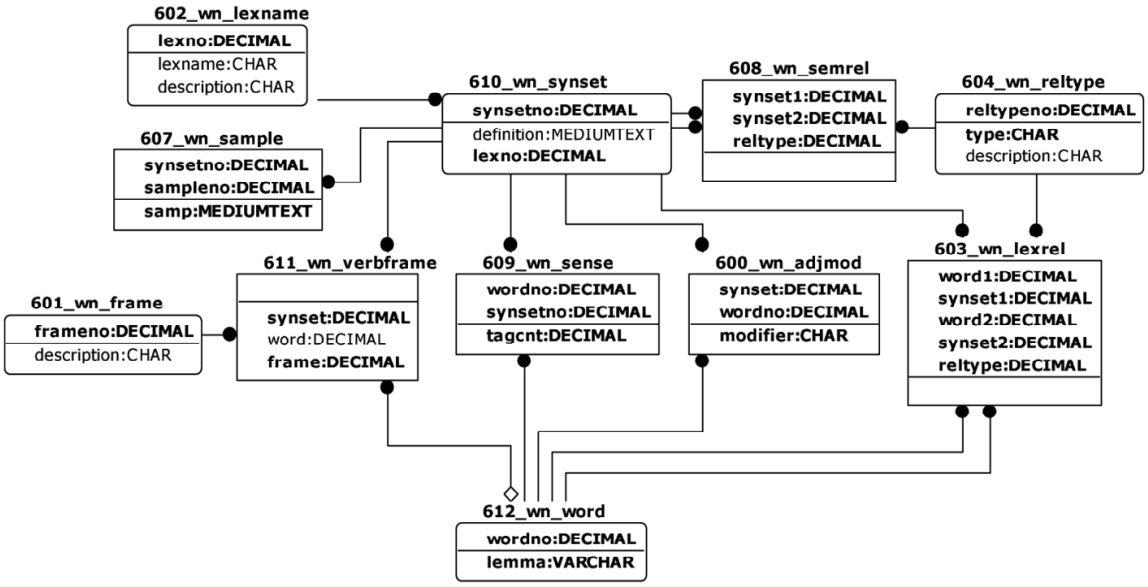


Figure 48: WordNet schema in various relations between synsets and senses.

In addition, semantic expansion over various relations (See Table 10 for details) will help a user to expand keywords to search for *wedding* and *matrimony* across coordinated synsets, or for *bridal* and *espousal* from its hyponyms.

8.5.1 Introduction of WordNet

To assess the above problems, we first need a common dictionary that a community of users will share for word indexing. For this purpose, we chose WordNet [47], which is a semantic lexicon for the English language. WordNet groups English open-class words into sets of synonyms called synsets that provide short definitions of words. An *open-class* word in linguistics is a word class that accepts the addition of items through such processes as compounding, derivation, coining, borrowing, etc. Hence, the lexical categories included in WordNet are nouns, verbs, adjectives, and adverbs.

WordNet currently has 0.2 million word-sense pairs. Figure 48 shows the complete schema representing the relationship between WordNet tables by which a user can query various relations between words. WordNet supports 28 relation types as listed in Table 10. Within those, recursive relation types can be used to trace related words up or down.

Table 10: WordNet synset relation types

Name	recursive	Name	recursive
hypernym	Y	hyponym	Y
instance hypernym	Y	instance hyponym	Y
part holonym	Y	part meronym	Y
member holonym	Y	member meronym	Y
substance holonym	Y	substance meronym	Y
entail	Y	cause	Y
antonym	N	similar	N
also	N	attribute	N
verb group	N	participle	N
pertainym	N	derivation	N
domain category	N	domain member category	N
domain region	N	domain member region	N
domain usage	N	domain member usage	N
domain	N	member	N

8.5.2 Semantic tags for c-data objects

To allow the semantic extension of c-data objects using WordNet, all WordNet dictionary data are converted into the relational database. Besides, their APIs are rewritten in server-side SQL statements to perform semantic tagging on c-data objects even without external applications including WordNet.

Figure 49 shows the schema of semantic tag data objects in ORM. Given one word to tag, its semantic meaning can be indexed in combination of words, synsets and lexical IDs. *WordNetID* is an identity of a set of three entities to tag a c-data object. This design permits multiple tags for one c-data object. Also each set is reusable by other c-data objects through the *TagID* table that works as a mapping table between c-data objects and WordNetID objects. This which reflects the fact that one data object may convey multiple semantic meanings. Figure 50 shows the relational schema implementation of Figure 49 in which a set of synset, word and lexical IDs uniquely identity the semantic meaning of a c-data object. Moreover this set is translated into 50 other languages. Thus, support for a query in a language other than English is feasible in this design. This tag structure permits multiple tags on one c-data object.

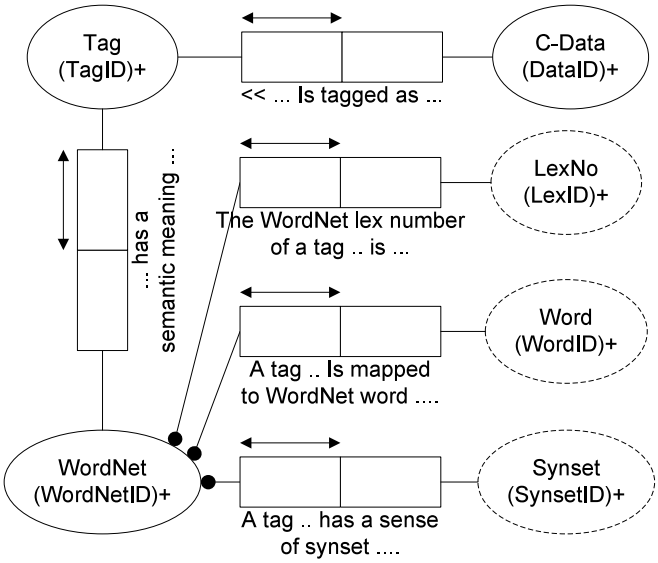


Figure 49: C-data object semantic annotation.

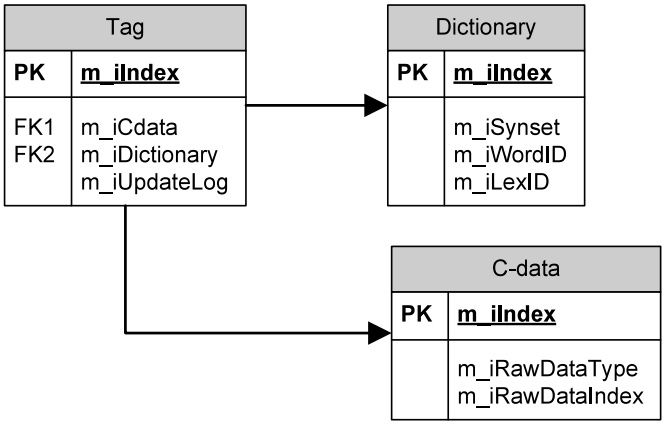


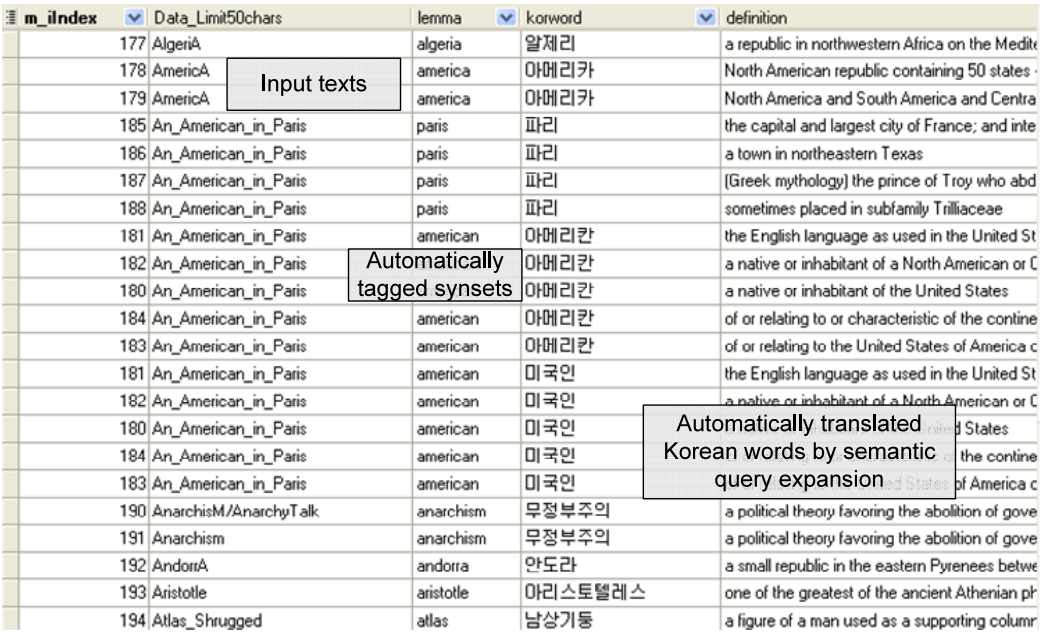
Figure 50: C-data object relational schema.

8.5.3 Internationalization for broad access

Using WordNet for semantic indexing [49] clarifies a word's correct semantic meaning and makes it feasible to exchange information over distributed networks. However, exchanging information written in different languages requires translation of each word that is to be exchanged. In this context, many efforts have been made for WordNet globalization [46, 97], and it currently has been translated into more than 50 languages (in 2009). Thus by through WordNet, users can exchange information in different languages.

The current E-model prototype supports such an automated translation environment using WordNet. Figure 51 shows an example in which the E-model system parses English texts into words and then automatically translates their synsets into matching foreign language synsets and associated words. This experiment translates some Wikipedia page titles located in the second column of the table. Titles are parsed into words in the third *lemma* column. And for each word, associated synsets are retrieved from English WordNet. Then we look at Korean WordNet, as an example, to find words that are associated synsets and shown in the fifth *definition* column. The forth *korword* column shows matching Korean words.

This example demonstrates that when the community shares a common dictionary, stored information can be translated into many other languages. Then a query can be written in any language that a system supports independent of its original language.



m_iIndex	Data_Limit50chars	lemma	korword	definition
177	AlgeriA	algeria	알제리	a republic in northwestern Africa on the Medit
178	AmericA	america	아메리카	North American republic containing 50 states
179	AmericA	america	아메리카	North America and South America and Centra
185	An_American_in_Paris	paris	파리	the capital and largest city of France; and inte
186	An_American_in_Paris	paris	파리	a town in northeastern Texas
187	An_American_in_Paris	paris	파리	(Greek mythology) the prince of Troy who abd
188	An_American_in_Paris	paris	파리	sometimes placed in subfamily Trilliaceae
181	An_American_in_Paris	american	아메리칸	the English language as used in the United St
182	An_American_in_Paris	american	아메리칸	a native or inhabitant of a North American or C
180	An_American_in_Paris	american	아메리칸	a native or inhabitant of the United States
184	An_American_in_Paris	american	아메리칸	of or relating to or characteristic of the contine
183	An_American_in_Paris	american	아메리칸	of or relating to the United States of America c
181	An_American_in_Paris	american	미국인	the English language as used in the United St
182	An_American_in_Paris	american	미국인	a native or inhabitant of a North American or C
180	An_American_in_Paris	american	미국인	d States
184	An_American_in_Paris	american	미국인	the contine
183	An_American_in_Paris	american	미국인	of America c
190	AnarchisM/AnarchyTalk	anarchism	무정부주의	a political theory favoring the abolition of gove
191	Anarchism	anarchism	무정부주의	a political theory favoring the abolition of gove
192	AndorrA	andorra	안도라	a small republic in the eastern Pyrenees betwe
193	Aristotle	aristotle	아리스토텔레스	one of the greatest of the ancient Athenian ph
194	Atlas_Shrugged	atlas	남상기둥	a figure of a man used as a supporting columr

Figure 51: C-data semantic tag view with automated Korean translations. (Special thanks to Sung-shin Im et al. at Pusan National University, South Korea for their Korean WordNet contributions.)

8.6 E-model prototype implementation

Based on [7], many graph data models have announced the implementation of their model. However, we could find no public codes or commercialized products that we could use to analyze, test, and compare with our E-model system. In fact, in contacting several of the authors of graph data models, we learned that their implementation ceased before public distribution. J. Hidders, who is the author of GDM [67], which incorporates representation of *n*-ary symmetric relationships, reported that GOOD [60, 48, 61] was the only model that actually has been implemented from his work. GOOD was known for manipulation as well as representations that are transparently graph-based with a sound theoretical basis. He mentioned that GOOD was implemented on top of a relational back-end and generates SQL statements for graph queries. But GOOD was developed in early 1990, and its developers did not open their work to the public nor did they publish their work on its implementation.

Efforts to implement graphs on a database have continued since then. Hybrid relational-XML databases [107, 53, 101, 17, 102] that we introduced earlier fall into this category in which tree-shaped XML documents can be parsed into a relational database. Even generic databases like MySQL have started supporting XML functions[1] that partially support XPath for tree-type queries. Oracle also adopted the RDF data model to support XML documents [3] and further expanded support for network-type graph data specialized for spatial or genome data analysis [128]. The common problem we see is that none of them introduced a generic type of data object that can be associated with other data models in a unified way. In other words, they added support for graph-structured data objects, but the relation between data objects is limited within a graph. If we want to add to some graph node more metadata in plain text or in some other data structure, such a relation is tricky and the structure of the source data object structure determines how the addition is made.

Our implementation departs from existing approaches in this context because we define

[1] http://dev.mysql.com/doc/refman/5.1/en/xml-functions.html

at the outset a new generic object called an e-node and then derive all necessary functions for a data storage system. Network-type data structures can be modeled naturally because of the system's rich relation support. In terms of implementation, hundreds of stored procedures and functions have been developed to support all the data object operations, their definitions, schema maintenance, and search constraints introduced in this work and to extend relational database functionalities to implement the E-model structure. Because all SQL server-side stored procedures and functions follow the SQL standard [42], any relational databases that conform to the SQL standard can embed the E-model database. This allows the full use of existing relational database systems and existing network supports.

Figure 52 depicts the relational E-model schema implemented on top of relational databases. This schema is composed of two main parts: (1) c-data objects and (2) e-node objects. C-data objects encapsulate raw data silos. Semantic tags are attached to c-data objects for extension. The RDAG schema objects are listed at the RDAG views.

Chapter 9 introduces how the E-model system can interoperate with existing data models. Based on numerous E-model prototype functions, Chapter 10 extends the SQL standard to support various E-model features to allow seamless connections between all data models stored in the database and provide a unified method to query such objects and their relations.

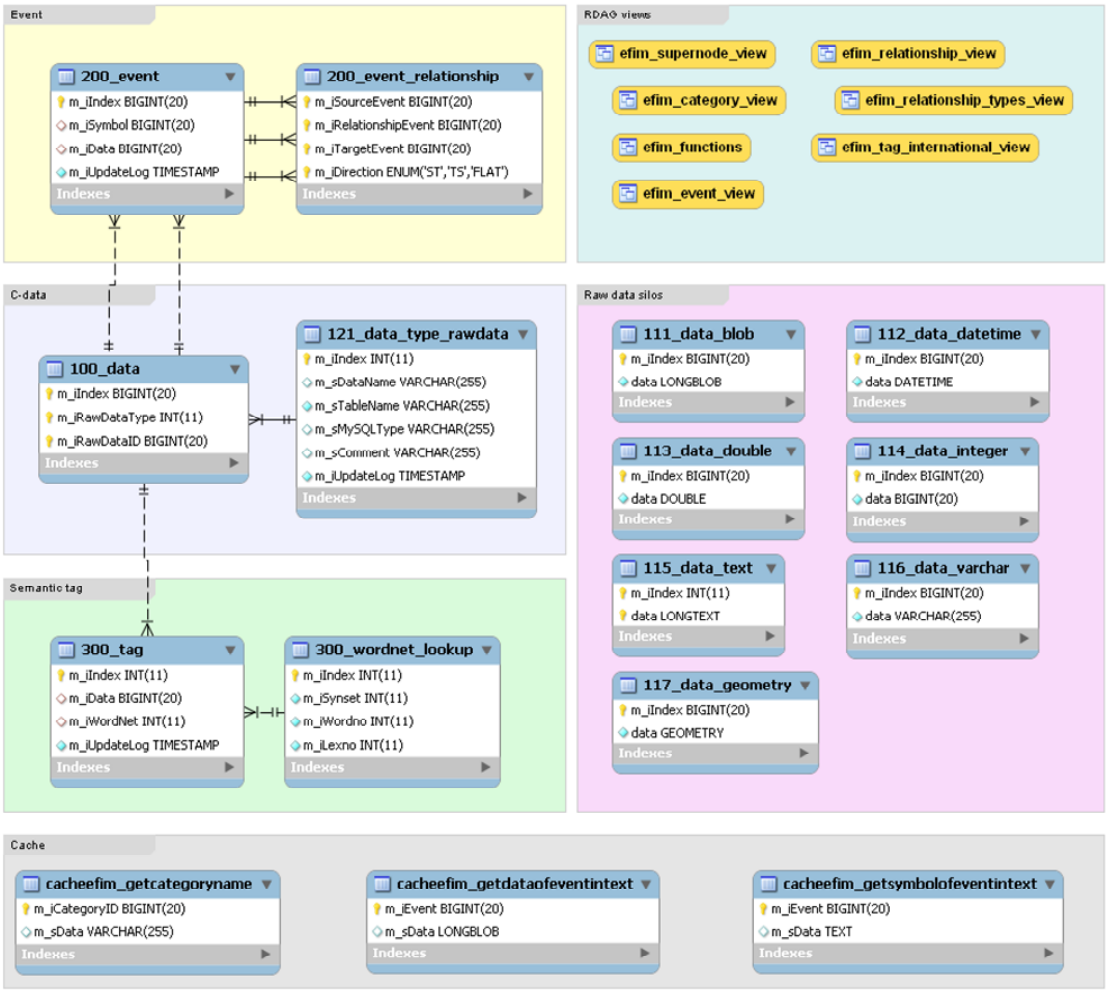


Figure 52: The E-model relational schema.

CHAPTER IX

INTEROPERATION WITH EXISTING DATA MODELS

When a new storage system is introduced, its interoperation with existing storage systems is a critical issue from many viewpoints. The E-model system by its graph-based architecture provides good flexibility to model disparate data models. This chapter details its interoperation with popular data models. We also handle the way to monitor other databases to maintain consistency and to investigate how to use the E-model system as a centralized database server.

9.1 Interoperation with the relational model

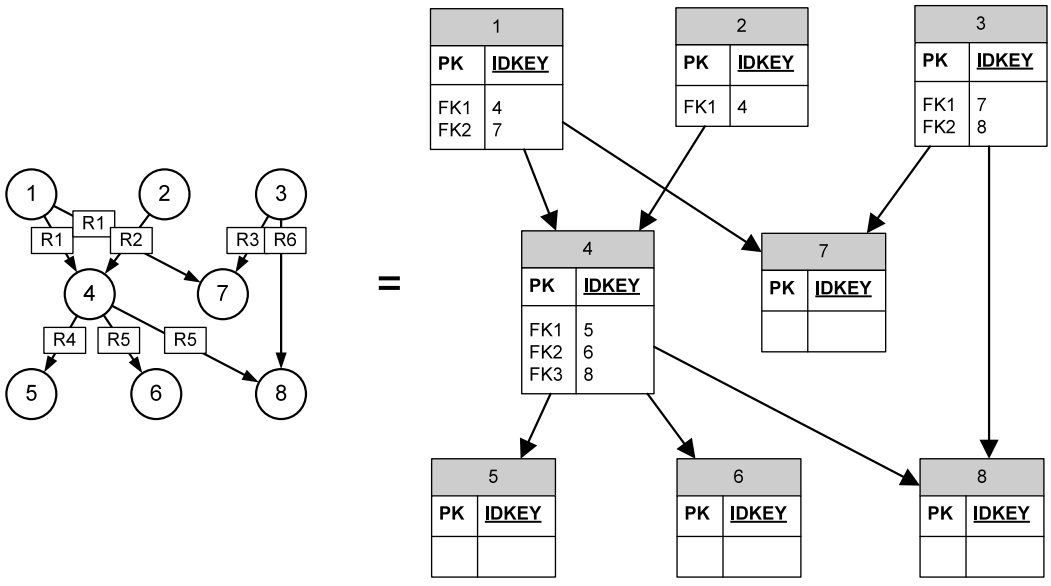


Figure 53: RDAG-to-relational model mapping example.

In relational databases, the relation between two tables is flat. Their semantic meanings, if they exist, are recorded as metadata as part of the comments in table definitions. Or those relations must be deduced from their column names, whereas in a RDAG, a relation is a

distinct e-node object. When it becomes flat (use of only one relation type), then a RDAG behaves like a sequence of first-order relations, which is similar to the relational models depicted in Figure 53. The difference, however, is that in a RDAG, the relation is assigned to the e-node object, whereas in relational models the relation is assigned between one specific type-dependent field in the source table and the other specific field of the target table. Because of this limitation, when a relational database is imported into the RDAG system, relations that it contains should be flat like a constant.

For an actual importing example, assume that we want to retrieve all three e-node sets $\{e_1, e_4, e_5\}$ having the relation R_1 in between $\{e_1, e_4\}$ and R_4 in between $\{e_4, e_5\}$. Let us note that the numbers of relational tables in Figure 53 represent the column names in an abstract way that is not permitted in the SQL standard. In a relational database such a query becomes two inner joins as shown in Query 9.1, and we cannot specify the type of their interrelation in a first-order query:

Query 9.1. Three-table relational JOIN example.

```
SELECT t1.IDKEY
FROM t1
INNER JOIN t4 ON (t4.IDKEY = t1.4)
INNER JOIN t5 ON (t5.IDKEY = t4.5)
```

Relational joins that appear in Query 9.1 can be represented more succinctly by the E-model. Besides, how data objects are queried differs from one data model to another. Assuming the interaction is the sequence of queries between disparate data models, let us first define an abstract query function based on the E-model.

Definition 9.1. An E-model query function, Q, is a recursive set of three entities returning e-nodes as the output. Q is prepared in a set composed of constraints (See Chapter 4) on source, relation and target e-nodes: $Q(e_s, e_r, e_t)$. All possible combinations and returning data structures are specified in Table 11. When Q returns e-node sets, then it can be

embedded within Q , forming a recursive query sequence.

Table 11: E-model query function format.

Format	Return type	Explanation
$Q(e_s)$	$\bigcup_{i=1,...,N}\{e_{r_i}, e_{t_i}\}$	Search all $\{e_r, e_t\}$ pair in any relation
$Q(e_t)$	$\bigcup_{i=1,...,N}\{e_{s_i}, e_{r_i}\}$	Search all $\{e_s, e_r\}$ pair in any relation
$Q(e_r)$	$\bigcup_{i=1,...,N}\{e_{s_i}, e_{r_i}\}$	Search all $\{e_s, e_t\}$ pair in e_r relation
$Q(e_s, e_r)$	$\bigcup_{i=1,...,N} e_{t_i}$	Search all target e-nodes from e_s in e_r relation
$Q(e_s, e_t)$	$\bigcup_{i=1,...,N} e_{r_i}$	Search all relations between e_s and e_t
$Q(e_r, e_t)$	$\bigcup_{i=1,...,N} e_{s_i}$	Search all source e-nodes from e_s in e_r relation
$Q(e_s, e_r, e_t)$	N	Return the count of the specified relation set

Query 9.1 using the Q expression is simply represented in a recursive query:

$$e_d :=> Q(R_1, Q(e_4, R_4, e_5)), \tag{41}$$

where e_d is a source e-node set to find.

9.1.1 Relational schema object interpretation

Data objects stored in the relational database have prefixed structures [42] in which multiple tables are related with others. This section suggests several design rules to parse relational schema objects so that the E-model can interoperate with relational database objects:

IR-1 The relational object schema forms a two-depth tree structure in a top-down order from *server* objects, *database* objects, and *table* objects. These objects match to E-model objects for each *server* to super e-nodes, *databases* to e-categories, and *tables* to child e-categories. Relations between objects for relational models are listed in Table 9.

IR-2 Existing keys and indexes are re-indexed when data instances are directly imported into E-model raw data tables that are fully normalized by design. A multi-column index, if it means something in addition to the unique constraint, can be modeled with additional relations between e-sdtypes.

IR-3 A foreign key in one relation table schema references the column of the other table. It is a directional reference from the source to the target table. In the E-model, the reference can be modeled by sharing the same e-sdtype between two e-categories. A direction from a source e-node to a target e-node can be modeled naturally by its e-group node registration time. For instance, when two e-group nodes share the same e-sdtype instance, the source e-group node should be registered before the target e-group node.

9.1.2 Interoperation with the a relational database

Instances populated in relational tables match e-group nodes in the E-model that are instances of e-categories. Let us first describe the algorithm to import relation data objects into E-model. Algorithm 7 assumes that the source data is specified with its associated database and table names. If the source is specified with a *SELECT* statement, then E-model first creates a view of the *SELECT* statement and reads the table metadata to register a matching e-category.

Algorithm 7 has several noteworthy features. Line 1 simplifies the hierarchical inheritance from a database object to an associated table with a concatenated string of database and table names. It creates a unique e-category that should be distinct within the system by concatenating the database and table name that makes it unique in the relational database. If a distinction between the database and the table is necessary, then a hierarchical inheritance from the database to the table can be feasibly modeled by inserting a child table e-category into the database e-category.

At line 2 in registering e-sdtypes, the set of a name and a raw data type of e-sdtypes should be unique in the E-model. If the foreign key in the relation table has the same field name of both the source and target table, then e-categories after the E-model import will share the same e-sdtype. Thus, it naturally supports a foreign key relation. However, if column names linked in the foreign constraint differ, which SQL allows, two approaches

Algorithm 7: Relational database table importing algorithm

Data: Database name D and table name T.
Data: Columns C of T and rows R of T
Data: Category e-node e_c, its e-sdtype e_{sd} and its e-function e_f.
Data: E-node name c-data ID c_n and e-node value c-data ID c_v.
Data: E-group node e_g.
begin
REM Create the e-category based on the table metadata.
1 $e_c \Leftarrow RegisterECatgory(\text{CONCAT}(D, ".", T))$
for $C_x \in C$ **do**
REM Register e-sdtypes for columns.
2 $e_{sd} \Leftarrow RegisterESDType(C_x.Name, C_x.RawDataType)$
REM Add new e-sdtype to e-category.
3 $AddESDTypeToECategory(e_c, e_{sd})$
REM Import raw data of each column C_x in T.
4 $ImportRawData(D, T, C_x.Name, C_x.RawDataType)$
REM Register e-nodes.
5 $c_n \Leftarrow GetCDataID(C_x.Name, \text{“}varchar\text{”})$
for $R_x \in R$ **do**
6 $c_v \Leftarrow GetCDataID(C_x.Value, C_x.RawDataType)$
7 $RegisterENode(c_n, c_v)$
REM Register e-node relations.
8 $e_{rr} \Leftarrow GetEFunction(\text{“}\_eRelation\_Row\text{”})$
9 $e_{rg} \Leftarrow GetEFunction(\text{“}\_eGroup\_Node\text{”})$
for $R_x \in R$ **do**
10 $e_g \Leftarrow CreateInstance(e_c)$
for $C_x \in C$ **do**
$RegisterRelation(e_g, e_{rr}, e_{\{C_x, R_x\}})$
$RegisterRelation(e_c, e_{rg}, e_g)$
end

can be applied: (1) Add an additional relation (*_eRelation_Foreignkey* in Table 9) between source e-nodes and target e-nodes, or (2) use a semantic tag to assign both column names the same semantic meaning.

Line 3 registers each column name as an e-sdtype, which is the child e-node of a table e-category. Line 4 imports all row data by column. Because all data objects in one column have the same name and same raw data type, column import is much faster than row insertion. When the row count exceeds the column count, this is in general a true statement. Line 5 retrieves c-data objects to register column names. Then it is used in a combination of registered column values to register new e-nodes. It should be noted that at this step duplicated name-value pairs are skipped and only new e-nodes will be registered to make the storage keep the first-normal form (1NF) that has no repetitive values. Lines 8 to 10 describe the mechanism to create e-group nodes that behave like the row ID of data objects for each record in the table. Finally, all e-groups are registered as child e-nodes of a table e-category.

Exporting instances of an e-category into the relation table is the reverse of the importing procedure as introduced in Algorithm 8. At line 1, from a given e-category, all associated e-sdtypes are used to build a column definition of a table to export. Next, from a given e-category, all e-group nodes, which are instances of an e-category, are retrieved because they work as the row instance identity of associated child e-nodes. Lines 7 to 8 fill up the table with all the child e-nodes of e-group nodes. Finally, line 9 drops the e-node index column that shapes the exported relational table to equal its original.

Algorithm 8: Relational database table exporting algorithm

Data: Category e-node e_c, its e-sdtype e_{sd}.
Data: Table schema statement S.
Data: Database name D, table T, columns C and identity column C_I.
Data: E-node name, c-data ID c_n, and e-node value, c-data ID c_v.
Data: E-group node e_g and all child elements of e_{gc}.
begin

REM Retrieve a set of child e-sdtypes of an e-category
1 $\{e_{sd}\} \Leftarrow GetListDecendants(e_c, GetEFunction(\text{“}\_eCategory\_Entity\text{”}), 1)$
REM Build the table schema statement from e-sdtypes
for $e_{sd} \in \{e_{sd}\}$ **do**
REM Retrieve each e-sdtype entities.
2 $e_{sd} \Leftarrow GetListDecendants(e_c, GetEFunction(\text{“}\_eSDType\_Entity\text{”}), 1)$
REM Add a column statement.
3 $S = AddColumnStatement(S, e_c, e_{sd})$
REM Create a table to back data.
4 $T \Leftarrow CreateTableFromSchemaDefinition(S)$
REM Retrieve all e-group instances of e_c.
5 $\{e_g\} \Leftarrow GetListDecendants(e_c, GetEFunction(\text{“}\_eGroup\_Node\text{”}), 1)$
REM Populate the table with e-group nodes.
6 $UpdateTable(T, C_I, e_g)$
REM Retrieve tuples of each e-group node.
7 $\{e_{gc}\} \Leftarrow GetListDecendants(\{e_{rg}\}, GetEFunction(\text{“}\_eRelation\_Row\text{”}), 1)$
REM Store the data back to the table by column in accordance with the matching e_g
for $C_x \in C\ of\ T$ **do**
8 $UpdateTable(T, C, e_g, e_{gc})$
9 $DropColumn(C_I)$
end

9.2 Interoperation with other data models using XML

In the case of relational databases, their structure can be derived directly from the metadata in tables in the database system because their design is based on a structured grammar. We developed the E-model XML schema to exchange data objects with those unstructured or semistructured data models whose structures are difficult to analyze or lack structural standards,. This section describes this feature in detail.

9.2.1 The E-model schema in XML representation

The XML schema [131] is standardized throughout the industry to unify the way XML document parsers interpret data objects in XML documents. Its path language XPath [15] and the query language XQuery [19] were developed based on the XML schema. However, there are many data models other than XML. Some of they have functional relations between entities that cannot be represented in XML documents or their structures are not fixed or standardized. Thus, an e-category XML schema that we develop in this section plays a central role in interpreting data objects from heterogeneous data models.

In our prototype design, we developed the E-model XML schema as depicted in Figure 54 (See Appendix. A for the complete schema with a sample XML document). This schema is an XML representation of an e-category model. Figure 54 resembles Figure 22, which is drawn in ORM. Any XML documents that conform to this e-category schema can register a new e-category, and the E-model system can parse their instances to import data objects from the source.

Let us note that in Figure 54, if we want to model an XML document, then *_eFunction* is unnecessary because the XML tree structure can be naturally represented by a child *_eCategory*. However, other data models, natural language parsers, or multimedia annotations may need to have a specific relation between *_eSDTypes*. Thus, *_eFunction* plays an important role in modeling the data structure of such entities into the E-model.

One may ask, why not directly interpret the XML schema to create a new *_eCategory*

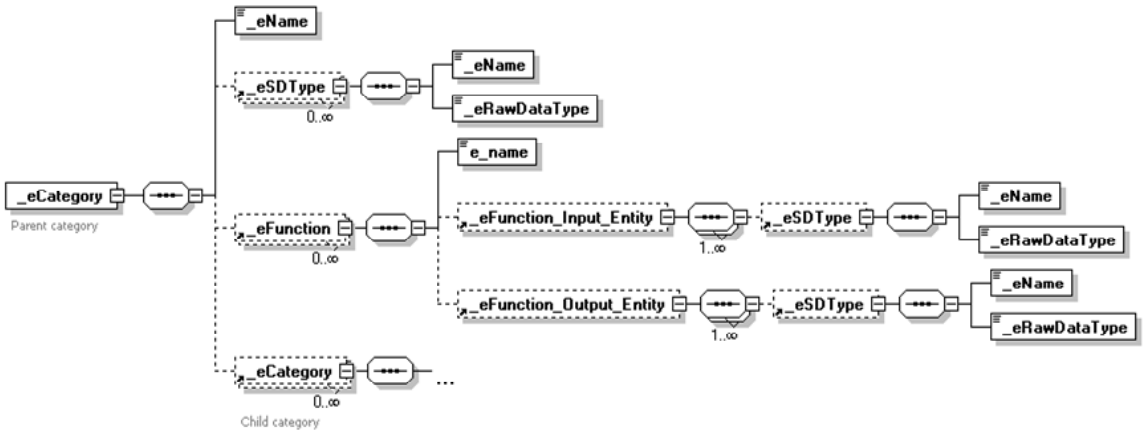


Figure 54: The E-model category in XML Schema.

As background knowledge, XML is often called a semistructured data model because of its rich text expression that lets both humans and computers interpret the structure of data objects and recognize the data values in them. For humans, XML's bracket separated elements and parent-child structure are easy to recognize as XML elements and analyze their hierarchical relations. For automated processing using computers, the XML parser uses the XML grammar schema [131] to define the XML structure and the element properties. The XML schema also helps in preparing the data input interface with various constraints on values; it also facilitates interpreting the data objects with their data types, value ranges, and references with other XML documents.

A simple XML document without a schema definition does not need to register an e-category and its elements. In such cases, the E-model simply creates an e-category without any child elements. It will add e-group instances interpreted directly from the source XML document. However, without a XML schema definition, the data type, correct existence, and repetitiveness of the data cannot be derived from the XML document alone. Therefore, before importing XML documents directly, the XML schema should be used to have the E-model system create an e-category to parse the XML document precisely. In this step, we need to consider several design approaches:

XD-1 The hierarchical structure of an XML document cannot be guessed exactly by the

XML document alone because the composition rule of each XML document may vary significantly. For instance in defining an XML element, if the *minOccurs* attribute is set as 0, then it may appear or disappear in the XML document. Besides, the raw data type of an XML element is specified in the XML schema - not in the XML document body. In the E-model system, knowledge of the type of raw data is a prerequisite to determining its raw data table.

XD-2 Depending on the back-end database, a system may support the XML parser or not. If the E-model is built on top of the relational back-end, then such an XML parser would not be available and the methods to handle XML documents are limited. Even in such a case, most modern relational databases support fundamental XML functions like *ExtractValue*, which supports an XPath query and the *UpdateXML* function that replaces specific elements in the XML document.

Thus we chose the option **XD-2** for users who want the E-model system be a stand-alone application without an external XML parser. Then an application solution is to use its own E-model XML schema that defines the structure of XML documents for interoperation. In this option, source XML documents should conform to the E-model XML schema.

9.2.2 Importing XML to E-model

When importing XML documents, their e-category schema is used to parse the XML body. The main steps in importing an XML document are described in Algorithm 9. Three main steps are involved: (1) find e-sdtypes of the current depth and register e-nodes, (2) find e-functions and create relations between matching e-nodes, and (3) if child e-categories exist, then shift down to the child level and recursively call this routine. More details are mentioned in the algorithm.

Algorithm 9: Import an XML document algorithm

Data: XML category name as D.
Data: Category e-node as e_c.
Data: Input XML document S and current input root node path as r_s.
Data: XML schema M and current root node path r_m of M.
Data: E-group node as e_g and instance e-nodes as e_s and data value as c_v.
begin

REM Get the e-category ID.
1 $e_c \Leftarrow GetECatgory(D)$.
REM Retrieve current level e-category model nodes in XML.
2 $r_m \Leftarrow GetListDescendantsXml(\{e_{rg}\}, GetEFunction(\text{“}\_eCategory\_Entity\text{”}), 1)$
REM Register the e-group node that will be the parent e-node of all child e-nodes at the current level.
3 $e_g \Leftarrow RegisterEGroupNode(e_c)$
REM Retrieve child e-sdtypes, e-functions and e-categories count from the schema.
4 $\{e_{sd}\} \Leftarrow RetrieveChildObjects(M, r_m, \text{“}\_eSDType\text{”})$.
5 $\{e_f\} \Leftarrow RetrieveChildObjects(M, r_m, \text{“}\_eFunction\text{”})$.
6 $\{e_c\} \Leftarrow RetrieveChildObjects(M, r_m, \text{“}\_eCategory\text{”})$.
7 $e_{rg} \Leftarrow GetEFunction(\text{“}\_eGroup\_Node\text{”})$
REM Register e-nodes.
for $e_{sd} \in \{e_{sd}\}$ **do**
REM Get matching XML elements and retrieve raw data values.
8 $c_v \Leftarrow RegisterRawData(S, r_s, e_{sd}.Name, e_{sd}.RawDataType)$.
REM Register e-nodes of each e-sdtype.
9 $e_s \Leftarrow RegisterENode(e_{sd}.Name, c_v)$
REM Register e-node relations.
10 $RegisterRelation(e_c, e_{rg}, e_g)$
REM Register relations between e-sdtypes.
for $e_f \in \{e_f\}$ **do**
REM Get matching e-nodes.
11 $\{e_{sd_{in}}\} \Leftarrow GetFunctionInputESDTypes(e_f)$.
12 $\{e_{sd_{out}}\} \Leftarrow GetFunctionOutputESDTypes(e_f)$.
13 $\{e_{s_{in}}\} \Leftarrow GetAssociatedENodes(\{e_s\}, e_{sd_{in}})$.
14 $\{e_{s_{out}}\} \Leftarrow GetAssociatedENodes(\{e_s\}, e_{sd_{out}})$.
REM Register relations between e-nodes.
15 $RegisterRelation(\{e_{s_{in}}\}, e_f, \{e_{s_{out}}\})$
REM Register child e-categories.
for $e_c \in \{e_c\}$ **do**
REM Shift XML root node.
16 $r_m \Leftarrow CONCAT(r_m, \text{“}\_eCategory[ID(e_c)]\text{”})$.
17 $r_s \Leftarrow CONCAT(r_s, e_c.Name)$.
REM Recursive call to register child e-nodes.
18 CALL $*this(e_c.Name, r_m, r_s)$
end

9.2.3 Exporting E-model to XML

E-model has labeled relations between e-nodes, whereas XML does not support a labeled relation. Thus, we need some design rules to govern whether to (1) select specific relations, or (2) ignore such relations and represent them all in a flat relation. Besides, the RDAG of the E-model forms an acyclic graph that can be considered a forest in comparison with the tree structure of an XML document. Thus, parsing a RDAG and associated E-model objects into XML documents needs rules on graph-to-tree decomposition. This section illustrates these rules with an example.

XML is an inherently tree-structured document. A graph without cycles is called an acyclic graph or a forest. A connected graph without cycles and with only one root node is a tree. Because a RDAG is an acyclic graph, it is naturally a forest.

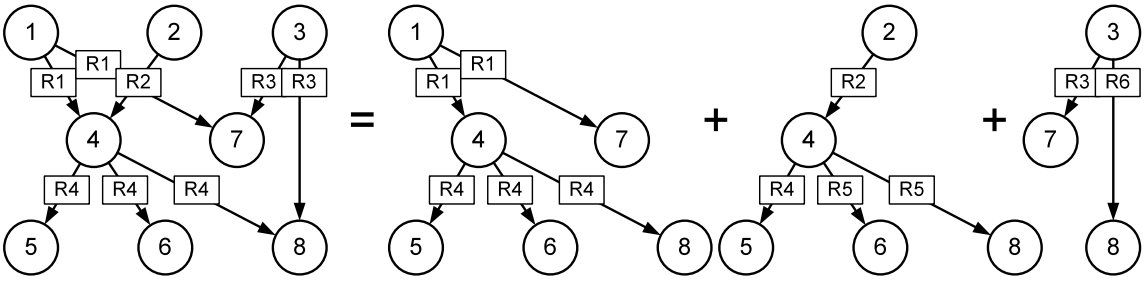


Figure 55: The DAG-to-tree decomposition.

Figure 55 shows a sample case of DAG-to-tree parsing. The root vertex of each tree is a node with only 1 outdegree. The leaf node has only 1 indegree. To parse the graph into trees when the relation between e-nodes is flat like a relational model or XML, we need prior knowledge of each node's identity. However, in a RDAG, which supports labeled relations between e-nodes, such tree decomposition can be specified simply by using the composition of relation e-nodes. This significantly reduces the complexity involved because the number of relation e-nodes is considerably fewer than the total number of e-nodes.

Let us practice this tree decomposition procedure with Figure 55. From the root e-nodes $\{e_1, e_2, e_3\}$, three trees can be decomposed using Q_r defined in Section 4.3. The first root

e-node (e_1) has child e-nodes by R_1 relation and subsequently has grandchild e-nodes by R_4 relation. In a constituent tree representation, this query can be formulated as:

$$T_1 :=> Q(e_1, R_1) \cap Q((e_1, R1), R_4 \cup R_5), \tag{42}$$

where T_1 is the first tree starting from e_1, Q is the relation query function returning the tree from the given e-node to the target e-node, and :=> indicates the postfix notation. Without relational specification, a query to parse the tree t_1 gets more complex as shown below.

$$T_1 :=> Q(e_1, e_4) \cap (Q(e_4, e_5) \cap Q(e_4, e_6) \cap Q(e_4, e_8)), \tag{43}$$

Queries on other trees are more simpler by use of the RDAG query expression:

$$\begin{aligned} T_2 &:=> Q((e_2, R_2), R_4), \\ T_3 &:=> Q(e_3, R_3). \end{aligned} \tag{44}$$

Finally a forest T can be represented with a set of three trees:

$$T := t_1 + t_2 + t_3 :=> Q(e_1, R_1) \cap Q((e_1, R1), R_4 \cap R_5) + Q((e_2, R_2), R_4) + Q(e_3, R_3). \tag{45}$$

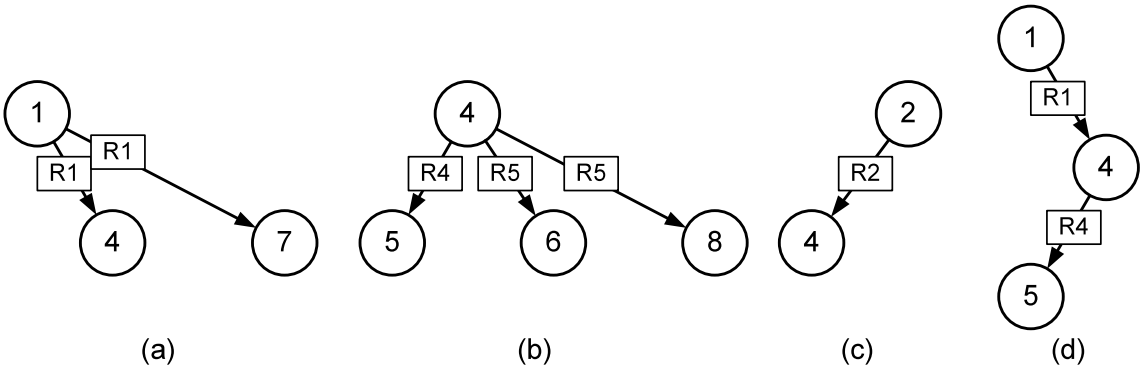


Figure 56: The RDAG subtree query examples.

This can be applied in the same way to retrieve subtrees using Eq. 46, and extracted subtrees are depicted in Figure 56.

$$\begin{aligned} T_a &:=> Q(e_1, R_1), \\ T_b &:=> Q(e_4, R_4 \cup R_5), \\ T_c &:=> Q(e_2, R_2), \\ T_d &:=> Q((e_1, R_1), R_4). \end{aligned} \tag{46}$$

As the above example clearly demonstrates, the tree query notation in a RDAG does not need to specify all child e-nodes to retrieve a tree from the forest. This is especially convenient when searching information in complex models without knowledge of their structures.

Exporting extracted trees into XML documents is straightforward; it first extracts e-node names and values and stores them into the bracket in hierarchical relations matched with that of a tree.

9.3 Interoperation with documents

The relation between e-nodes and c-data objects may act as an inverted list, which makes the E-model a candidate for a Web search back-end storage. An inverted list [65, 137, 36] is a popular storage index for Web search engines. This section handles a part of E-model features in this context that may contribute to Web search applications.

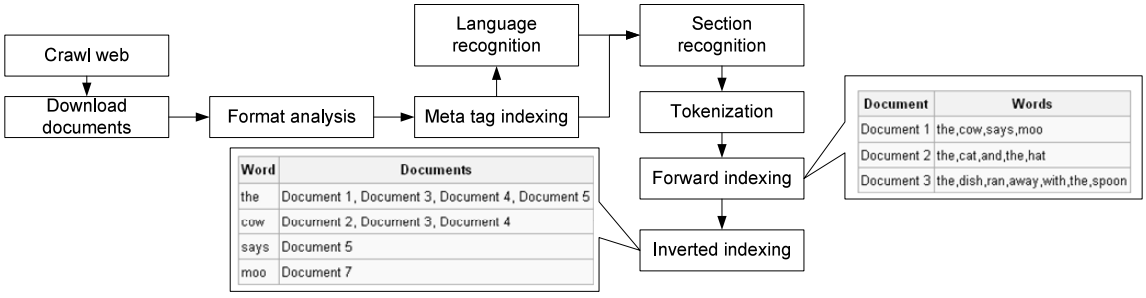


Figure 57: Web document indexing flow example.

Figure 57 shows a typical Web document indexing flow. (For details on the Web search mechanism, please read [23, 90].) A Web crawler from the search engine navigates the

Web to collect documents. The type of document is not limited to HTML but can be in various formats such as XML, PPT, DOC, PDF, and many others. A content structure thus depends on the document format, and the technology necessary for content analysis should be developed for each of them. Once the content to build indexes is successfully collected, the meta tags and languages of this content are first identified. For text content represented in various layouts, section localization is necessary to separate the content into different roles before we parse them into tokens. When all words are finally extracted from the documents, forward and inverted indexes are built for data access. A sample table of each index type is attached in a box on each side of Figure 57.

The inverted list [65, 137, 36] then can return a query result directly from the user's search keyword. From an example in the figure, when a surfer asked "cow," then the inverted list will return results: "[Document 1, Document 2, Document 3]" in a one-time lookup on the list. An inverted list is thus an index data structure that directly maps content to its storage locations in the database system. By using the inverted list, user queries in the combination of keywords can be mapped to documents that include words of interest. Hence, a full text search can be replaced by the inverted list without necessarily looking for all Web pages, an effort that requires significant time.

Assuming the procedures in Figure 57 are developed and performed on top of the E-model back-end, we can expect several benefits compared with conventional inverted lists.

II-1 In the E-model, one-to-one relations between e-nodes and c-data objects work just like that of an inverted list. This is done by performing a query on c-data objects first and then retrieving associated e-nodes with their siblings under the same e-group node. In addition, this approach in the E-model supports a unified access, which is independent of raw data types, document types, or their complex relations.

II-2 Various search constraints introduced in Chapter 4 can be applied. Thus, for raw data objects other than text-type objects, value evaluation, and range selection is very efficient. Various search constraints introduced at Chapter 4 can be applied.

9.4 Materialized views for E-model

The E-model supports complex modeling methods, and it is also feasible for the E-model to be exported in the form of popular data models. For users who prefer to keep their existing database communication methods, the E-model system may act in two ways: (1) as a stand-alone database server with materialized views and (2) as a back-end database for all other databases and work only for E-model specific queries. We will handle the second topic in Chapter 10. This section will focus on stand-alone capability.

A materialized view caches the query result as a concrete table that may be updated from time to time from the original base tables. This enables much more efficient access compared with a virtual table view, which builds results online at the potential cost of some data being out-of-date. This type of operation is most useful in data warehousing scenarios in which frequent queries of the actual base tables can be extremely expensive. For instance, if a user connects to the E-model system through the ODBC interface, then the E-model system may forward the query to the materialized view of the internal data model. A materialized view in the E-model is not limited to a table but also can be represented for disparate data models like XML. It can be further expanded to other applications such as a FILE system because the File Allocation Table (FAT) is a simple directed graph.

Materialized views can be refreshed in real time when any updates or insertions occur at the associated e-category. And the update rule that requires maintaining the latest state of the data is handy in the E-model system compared with other approaches. For instance, the Oracle materialized view [14] logs the access to the database system to monitor UPDATE- or INSERT-like commands that alter the status of the materialized view. If the status is changed, then Oracle rebuilds the materialized view on request.

Such additional burdens that monitor and analyze all requests to the database server are not necessary in the E-model system. A RDAG has connected group nodes in a temporally ordered manner. Thus, when a request for a materialized view arrives, the E-model checks and compares the most recent e-group nodes of the associated category nodes with

the materialized view to determine whether to update the view. Since each e-node has a transaction timestamp (See Section 3.4), such a comparison is a straightforward computation of the temporal distance between two e-nodes: (1) the most recent e-group node of the E-model system and (2) the latest e-node used in building the materialized view.

The reason that the Oracle materialized view has to monitor all activities is because its *UPDATE* command does not increase the size of data but simply replaces the data. So the record count or data size that does not require online monitoring does not work in the case of Oracle. However, in the E-model system, a newly inserted e-node for *UPDATE* has a new transactional timestamp. So our rule holds for *UPDATE* and *INSERT* because both commands affect the status of the database.

This approach makes it feasible to use the E-model system as a proxy database server for external access, while keeping the E-model transparent to the client system as if the E-model server were behaving exactly like other database systems.

The use of the materialized view makes the E-model system act as a centralized archive as well as a transparent proxy database server depicted in Figure 58. This system design uses the superior flexibility of the E-model design as an inter-medium between the information sources and other information systems while providing equivalent query performance with existing databases.

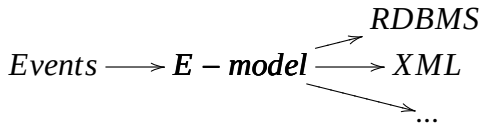


Figure 58: The role of E-model as an inter-medium storage.

9.5 Monitoring database activities

This section introduces a case in which the E-model system coexists with existing databases. Based on the interoperation algorithms introduced before, a E-model system can monitor the activities of other online databases to catch events of interest to mirror their instances. In

this configuration, queries can be applied to both the E-model and other database systems. For instance, an unstructured query will be forwarded to the E-model system for information retrieval, while other queries for structured data models are forwarded to the existing database. This idea is further extended in Chapter 10 to a new database language named EML that supports both structured and unstructured queries with various SQL extensions.

This section illustrates ways for the E-model system to monitor relational database cases. In a relational database, three types of constraints can be defined for triggers: IN-*SERT*, *DELETE*, and *UPDATE*. A database trigger[1] is a procedural code that is automatically executed in response to certain events on a particular table in a database. There are two classes of triggers; they are either "row triggers" or "statement triggers." Row triggers define an action for every row of a table, while statement triggers occur only once per *INSERT*, *UPDATE*, or *DELETE* statements. This section will deal with statement trigger cases.

For other data models like XML, let us assume they employed XML-enabled rela-tional databases as their back-end. Based on the work in this section, similar techniques can be developed for proprietary databases.

9.5.1 Monitoring INSERT operations

This section explains the algorithm that installs the *INSERT* operation monitoring trigger to update the E-model database. Algorithm 10 shows the overall flow that first registers an e-group node as the identity of all tuples in the inserted record. Each tuple creates an e-node and maps it back to the e-group node.

Let us set up one simple example for a clear understanding. The table below is the table definition of Wikipedia's *hitcount* table. It includes one integer column.

Example 9.1. A Wikipedia *hitcounter* table definition:

```
CREATE TABLE ‘hitcounter‘ (
```

[1] http://en.wikipedia.org/wiki/Trigger_(database)

Algorithm 10: Monitor relational database INSERT operation

Data: Source database as D.
Data: Source table as T.
Data: Source e-category name as C.
Data: Target E-model database name as E.
Data: A new e-group ID as e_g.
Data: e-super node ID as e_s.
Data: Prepared statements as S.
Data: Column name c-data ID as I_c.
Data: Column raw data type ID as I_d.
begin

REM Register a new e-category for the relational table.
1 $e_c \Leftarrow RegisterRelationalECatgory(C)$.
REM Set constant values.
2 $r_d \Leftarrow GetEFunction($"_eCategory_Entity"$)$
3 $r_s \Leftarrow GetEFunction($"_eSuper_Relation"$)$
REM Prepare create trigger statements.
4 $S \Leftarrow$ CONCAT("CREATE TRIGGER _eTrInsert", T)
5 $S \Leftarrow$ CONCAT(S, "AFTER INSERT ON ", D, ".", T)
6 $S \Leftarrow$ CONCAT(S, ...) // Auxiliary specifications are omitted
REM Set CURSOR to retrieve column names and raw data types from INFORMATION_SCHEMA.
for $c_i \in \{C\}$ **do**
REM Set constant values.
7 $I_c \Leftarrow GetColumnRawDataTypeID(c_i)$
8 $I_d \Leftarrow GetColumnName(c_i)$
REM Register an e-node for each tuple with its relation with the e-group node.
9 $S \Leftarrow$ CONCAT(S, "$RegisterRelation($", e_g, ",", r_d, ", $RegisterENode($", I_d, ",", I_c, ", $GetColumnValue($", c_i,")")
REM Run the statement to register a trigger.
10 PREPARE run_stmt FROM S; EXECUTE run_stmt;

end

```
  `hc_id` int(10) unsigned NOT NULL
);
```

Based on Algorithm 10, the E-model system installs a trigger as shown in the trigger definition in Figure 59. As shown in the figure, our algorithm minimizes access to the E-model system by replacing fixed values with constant values. If a source table has additional columns, then accordingly data, event and relation registration codes will be repeated.

9.5.2 Monitoring DELETE operations

The *DELETE* operation must consider several options as discussed in Section 6.2. Deleting a record in a relational database means removal of one e-group node. For simplicity, Figure 59 shows one design case that because they can be referenced in the future, does not remove child e-nodes.

9.5.3 Monitoring UPDATE operations

The *UPDATE* operation can be understood as the sequence of *INSERT* and *DELETE*. To maintain data validity, *INSERT* should be performed first and then *DELETE* should follow. If we want to retain the history of changes (for an archive-type engine), then the E-model trigger *FOR UPDATE* would be equal to the *INSERT* operation that considers the instance *FOR UPDATE* operation as a new insertion.

```
CREATE TRIGGER `efim_tr_insert_hitcounter`
  AFTER INSERT ON `hitcounter`
  FOR EACH ROW
BEGIN

 DECLARE iCategoryID BIGINT(20) UNSIGNED DEFAULT NULL;
 DECLARE iGroupID BIGINT(20) UNSIGNED DEFAULT NULL;
 DECLARE iEventID BIGINT(20) UNSIGNED DEFAULT NULL;
 DECLARE iEventRelationshipID BIGINT(20) UNSIGNED DEFAULT NULL;
 DECLARE iSymbolID BIGINT(20) UNSIGNED DEFAULT NULL;
 DECLARE iDataID BIGINT(20) UNSIGNED DEFAULT NULL;
 DECLARE iRawDataTypeID BIGINT(20) UNSIGNED DEFAULT NULL;

 SET iCategoryID = 114;
 SET iSymbolID = 151;
 SET iRawDataTypeID = 6;
 SET iGroupID = wiki.EFIM_PutGroupEvent(
                'wiki.hitcounter');
 SET iEventRelationshipID = 6;

 INSERT IGNORE INTO emodel.200_event_relationship(
        m_iSourceEvent, m_iRelationshipEvent, m_iTargetEvent)
 VALUES (iCategoryID, iEventRelationshipID, iGroupID);

 SET iEventRelationshipID = 12;
 INSERT IGNORE INTO emodel.200_event_relationship(
        m_iSourceEvent, m_iRelationshipEvent, m_iTargetEvent)
 VALUES (1, iEventRelationshipID, iGroupID);

 SET iEventRelationshipID = 6;

 SET iRawDataTypeID = 4;
 SET iDataID = emodel.EFIM_PutData(4, new.hc_id);

 SET iSymbolID = 153;
 SET iEventID = emodel.EFIM_PutEvent(iSymbolID, iDataID);

 SET iEventRelationshipID = 8;
 INSERT IGNORE INTO emodel.200_event_relationship(
        m_iSourceEvent, m_iRelationshipEvent, m_iTargetEvent)
 VALUES (iGroupID, iEventRelationshipID, iEventID);

END;
```

Figure 59: Wikipedia Hitcount table INSERT trigger definition.

```
CREATE TRIGGER ‘efim_tr_delete_hitcounter‘
  AFTER DELETE ON ‘hitcounter‘
  FOR EACH ROW
BEGIN

 DECLARE iGroupNode BIGINT(20) UNSIGNED DEFAULT NULL;

 SET iGroupNode = (
        SELECT DISTINCT r1.S
        FROM
          emodel.efim_relationship_view AS r1
        WHERE
          (r1.TN = 'hc_id' AND r1.TD = old.hc_id)
          AND r1.SN = 'wiki.hitcounter' LIMIT 1
 );

 DELETE FROM emodel.200_event_relationship
 WHERE m_iTargetEvent = iGroupNode OR m_iSourceEvent = iGroupNode;

 DELETE FROM emodel.200_event
 WHERE m_iIndex = iGroupNode;

END;
```

Figure 60: Wikipedia Hitcount table DELETE trigger definition.

CHAPTER X

THE E-MODEL LANGUAGE: EML

EML is an E-model data query language extending SQL [26] with enhanced features that include but are not limited to: (1) unstructured query, (2) semantic query expansion, (3) temporal query, (4) ranked ordering, (5) path query, and (6) natural join. Section 10.1 explains the system configuration and environments in which EML mostly works well. Section 10.2 introduces the features of EML with rich examples. Section 10.3 formally defines the EML language in BNF form. Section 10.4 handles topics of EML translator implementation.

10.1 E-model system configuration

This section illustrates the difference between structured and unstructured query environments. We will demonstrate the role of the E-model system as the centralized proxy database server to solve issues related to a given example.

Figure 61 depicts a multimedia information query environment. It simulates a complex database system configuration from a given picture identifying a person to retrieve related information. Directional lines connecting objects in the figure represent the flow of queries over disparate databases.

We assume the role of each database as follows; DB1 stores pattern databases to detect humans within a picture. DB2 is a collection of personal features to identify a person. Using a person's name as its identity, DB3 supports a relational query about the person's job description. DB4 supports a hierarchical query about his family tree. Other databases up to DB[N] are independent personal information databases specialized for each person to search. Each database may contain multiple structured schema.

In Figure 61, each query goes exactly to one specified database, which means a query

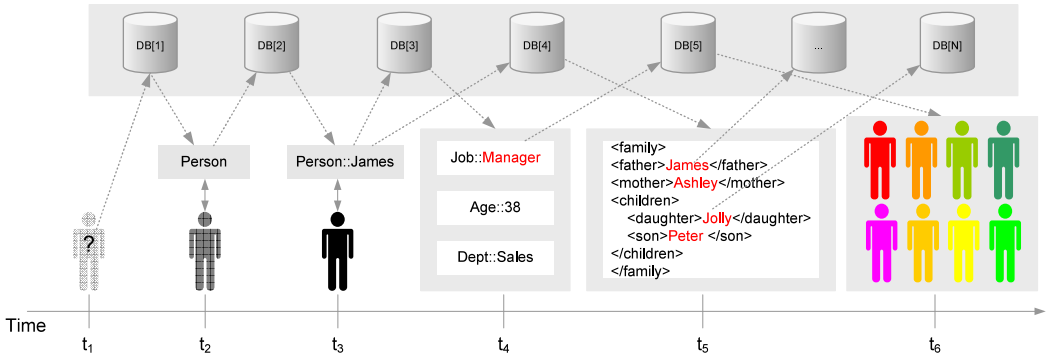


Figure 61: A completely structured query system configuration.

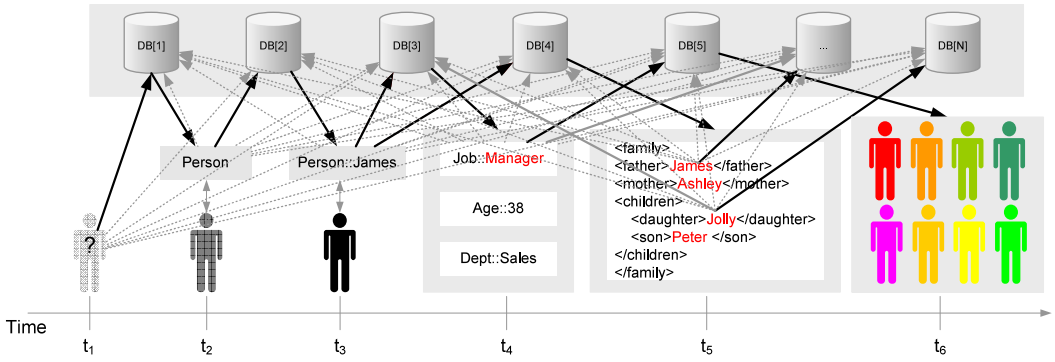


Figure 62: An unstructured query system configuration.

statement is prepared exactly for the target database. In this environment, the complete structure and schema of each database are assumed to be known a priori, and each query statement is prepared separately for each disparate database using each database's own query language (ex. SQL for relational databases; XQuery for XML documents, Text query for unstructured documents). This configuration may provide the most efficient query process-ing environment. However, management of disparate schema and their interrelations with other databases have high maintenance requirements. This example includes personal information databases in which associated schemas and type of instances may significantly vary from person to person. If we need to manage a large number of members, then in practice ever an overhead to supports a structured query for such environments ever grows up.

If a target database is not known, but the type or value of objects of interest is known,

then a schema-less query (also called an unstructured query) should be applied to all databases as illustrated in Figure 62. In doing so, each query should be prepared in multiple languages for each database. This method significantly lowers the query performance with high complexity.

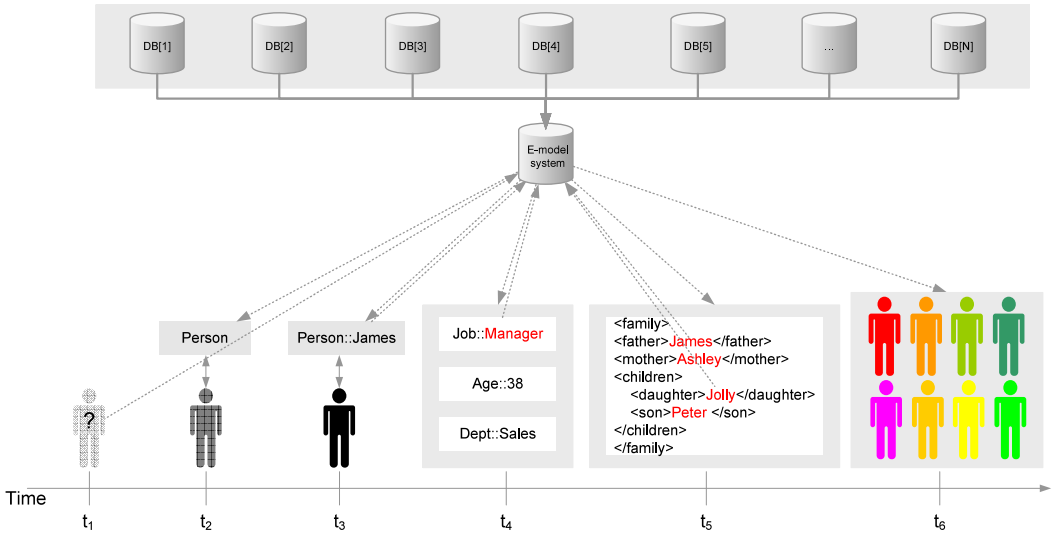


Figure 63: The E-model proxy database server configuration.

Figure 63 recommends a database system configuration in which the E-model database acts as a centralized query proxy server that enables unified access to disparate databases. Technical implementation of this system configuration has already been discussed in Chapter 9. In fact, the E-model system can work in three modes:

MD-1 An independent stand-alone database system with materialized views for disparate data model access (See Section 9.4).

MD-2 A back-end database system that logs all instances of disparate databases exclusively for unstructured information retrieval (See Section 9.5).

MD-3 A proxy database server that provides a unified data query language for disparate databases with extended features.

For MD-1, all existing databases, their schema, and instances are imported into the E-model system, and then a user accesses only the E-model system. For MD-2, a user sends a query to a specific database when its schema is known. If the schema is unknown, the user sends a query to the E-model system to retrieve information from disparate databases. In the case of MD-3, a new query language should be introduced. As a proxy server, it should forward structured queries to the target database. For queries that use E-model features, it should translate the statement so as to perform the expected operations. This chapter introduces a new E-model language called EML with details on its roles and features.

10.2 Extended SQL

An E-model prototype is implemented on top of a relational database. Its full functions are implemented using hundreds of server-side SQL-stored procedures and functions. This means that the E-model system can interoperate with existing databases under the same configuration. It can use the same communication protocol, same server, and client interfaces without changes. Thus, additional overhead to maintain an independent database system is not required.

In this configuration, the E-model system can provide many benefits based on the notable features developed in this work. This section illustrates ideas for implementation of a new language that we have named E-model language (EML) to further extend SQL. EML is in an early stage and will be extended beyond its introductory discussion in this section. Table 12 lists newly introduced notations to extend the SQL namespace, and SQL expanded functions are listed in Table 13.

In Table 12, * represents all sets for an unstructured query, which means the e-nodes to search. Thus, e-node predicates that project a e-node property like its name, value, type, or timestamp can be applied. Relations between e-nodes are specified by the relation operator in Table 13. Ordered parent and child object expressions represent the temporally ordered e-node relations. For many-to-many connections, the order represents the historical

Table 12: The EML namespace.

Name	Notation	Comments
All sets	*	Search all objects in the E-model database.
Parent object	_parent, _parent[i]	*i* th parent object.
Child object	_child, _child[i]	*i* th child object.
Name	_name	Object name.
Value	_value	Object value.
Raw data type	_type	Object raw data type.
Time	_timestamp	Object transaction registration time.
Temporal search	WHEN	Temporal search of e-node time predicates.

connection in which the oldest e-node is the first object.

10.2.1 Unstructured query

An unstructured query means a query without any specification of the data structure to search. E-model constraints introduced in Chapter 4 support such an unstructured query. Query 10.1 shows an unstructured query example to search over all databases to find any type of data objects of interest. Internally, the E-model searches over e-nodes with "911" value. It should be noted that this query does not specify the data schema to search, and it does not even specify the field name of that value. A Web search using a keyword exactly matches with this query statement.

Query 10.1. Search all data objects that have a value "911".

```
SELECT *
FROM _* AS t1
WHERE t1._value = "911";
```

10.2.2 Semantic expansion

The second example specifies the name and value of data objects to query. Constraints on the text comparison functions can be expanded by semantic operators as defined in Table 13.

Query 10.2. Search data objects named like “memo” synonyms and whose value includes the word “party.”

```
SELECT *
FROM _* AS t1
WHERE t1._name LIKE _SYNS("memo") AND
      t1._value REGEXP "party";
```

10.2.3 EML namespace example

The third example specifies the type of data objects in addition to the name and value of objects. This query sets constraints on e-nodes properties.

Query 10.3. Unstructured query extension from Example 4.6.

```
SELECT *
FROM _* AS t1
WHERE t1._name LIKE "episode" AND
      t1._type = "INTEGER" AND
      t1._value = 601;
```

10.2.4 Temporal query

By the very nature of e-nodes, the E-model supports temporal queries based on transaction time. A temporal search also exists in SQL for a valid time field. In the E-model, all e-nodes embed a transaction timestamp, thus, EML can support an independent temporal query using *WHEN* as a constraint on the temporal ranges of e-nodes.

Query 10.4. Find “BLOB” type data objects registered yesterday.

```
SELECT *
FROM _* AS t1
WHERE t1._type = "BLOB"
WHEN Day(Now()) - 2 day < Day(t1._timestamp) <= Day(Now()) - 1 day;
```

10.2.5 Ranking queries

The E-model can prioritize the order of objects by their popularity (See Section 4.4). This is useful in information retrieval for ordering results by their reference count. Also, this method works for any e-nodes because any e-nodes joined in the relation table will have a reference count. This means *POPULARITY* always works in any query statement.

Query 10.5. Select the most popular 100 data objects.

```
SELECT *
FROM _* AS t1
ORDER BY _POPULARITY(t1)
LIMIT 100;
```

10.2.6 Path query

The relations between e-nodes are stored in a three-tuple adjacency list (See Section 7.6.1). Thus, constraints on relations are applied only to the middle tuple of the list. Also, the types of relations are finite and assumed to be known a priori. Thus, a path query composed of names of concatenated e-functions specifies a path in the adjacency list. It is a fast and succinct constraint as demonstrated in the EML-to-XML exporting process (See Section 9.2.3). The examples below select objects of a subject “Jason” in sentences.

Query 10.6. Select verbs and objects for a subject “Jason” from sentences in any data objects.

```
SELECT *,
       _R(%1, "_eSubjectVerb/_eVerbObject", *)
FROM _* AS t1
WHERE t1._name LIKE "%lemma%" AND
      t1._value = "Jason";
```

Query 10.7. Rephrase Query 10.6 using in recursive path form.

```
SELECT *,
       _R(%1, "_eSubjectVerb", R(*, "_eVerbObject", *))
FROM _* AS t1
WHERE t1._name LIKE "%word%" AND
      t1._value = "Jason";
```

10.2.7 Schema object manipulation

In the E-model, e-sdtypes are distinct objects and shared by e-categories. This is like a constraint in relational database, a field name should be distinct within a database system. Thus if a user wants to remove some column for all databases, he can simply drop one e-sdtype node. Then EML will drop linked relations.

Query 10.8. Drop the "SSN" column from all tables.

```
ALTER TABLE _* DROP COLUMN SSN
```

10.3 EML grammar

A computer language to be parsed by a compiler or an interpreter should have a formalized syntax that conforms to some standard that a general parser can support. In our work, an EML grammar is prepared in Backus-Naur Form (BNF) [81]. BNF is a metasyntax used to express context-free grammars using two sets of rules: i.e., lexical rules and syntactic rules. BNF is widely used as a notation for the grammars of computer programming languages, instruction sets, and communication protocols as well as a notation for representing parts of natural language grammars. Many textbooks on programming language theory and/or semantics document the programming language in BNF because once a language is written in BNF, its compiler or interpreter implementation is assuredly feasible.

The current BNF of EML grammar is listed in Appendix B. The EML BNF is an extension of an ad hoc version of SQL 89 (X3.135-1989, ISO/IEC 9075::1989) for Gold Parser [34]. It currently has definitions for the most fundamental operations, including

SELECT, INSERT, UPDATE, CREATE, ALTER, and DROP SQL statements that are included in the SQL-89 standard. We built the grammar using Gold Parser[1], which is a parsing system that uses the LALR (Lookahead Left-to-Right) algorithm [2] to analyze syntax and a deterministic finite automaton [126] to identify different lexical units.

10.4 EML translator

An EML translator is a front-end user query processor that converts EML statements into a set of codes to run on the E-model system. Because the first E-model prototype system has been developed atop a relational database system, all functions (named E-model SQL APIs) of the E-model system are server-side stored procedures and functions. Thus, the role of the EML translator is to convert EML statements into a set of SQL statements that call E-model SQL APIs. Because the EML translator converts EML statements into complete SQL statements that use existing protocols to interact with the database, it can be a stand-alone application for the client or it can be a part of the server-side E-model system.

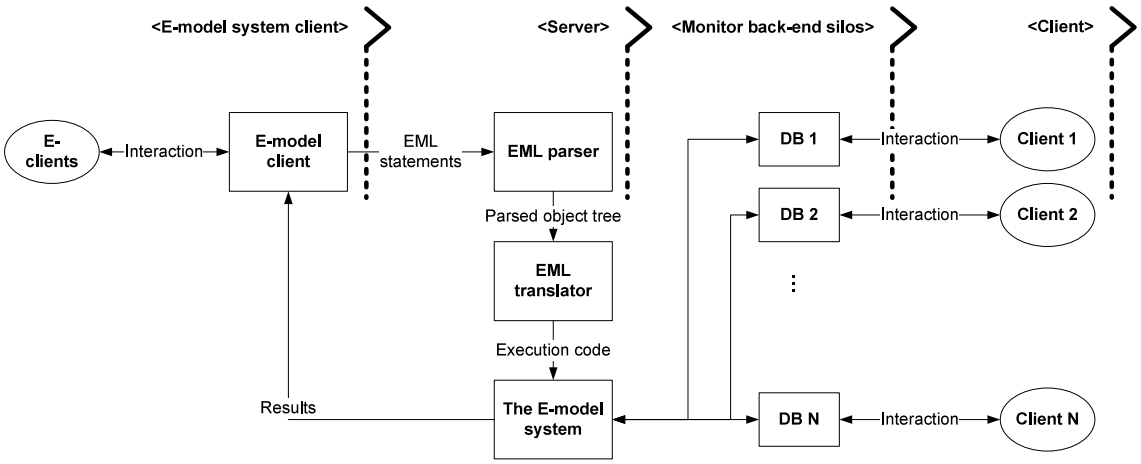


Figure 64: The E-model system computation flow.

Figure 64 illustrates the computing flow within the E-model system in the MD-3 configuration (See Section 10.1). The E-model system monitors and updates its companion

[1] http://www.devincook.com/goldparser/index.htm

databases, and the query can be issued to the E-model and also to each database. This section will explain the left part of the figure, which assumes a client for the E-model system (named E-clients) issues the EML statement to the system. Each EML statement is first parsed on the basis of the EML BNF grammar and then translated into the SQL statement to run on the E-model system. A translator also has a role in output manipulation. For the details, let us use an example to explain this procedure.

10.4.1 Parsing example

Query 10.9 below is an unstructured query to search all data objects having an *Integer* type value *601*.

Query 10.9. Find all data objects having "601" as their value.

```
SELECT t1._name, t1._value, t1._timestamp
FROM _* AS t1
WHERE t1._type = "Integer" AND t1._value = "601";
```

The current E-model prototype has numerous SQL stored procedures and functions to provide complete data operations. Based on those, Query 10.10 is the translated SQL statement that calls E-model SQL APIs in which *EFIM* is the prefix for API function calls.

Query 10.10. Converted SQL statements using E-model SQL APIs.

```
SELECT 200_event.m_iIndex,
  EFIM_GetDatainText(200_event.m_iSymbol) AS name,
  EFIM_GetDatainText(200_event.m_iData) AS data,
  200_event.m_iUpdateLog
FROM 200_event
WHERE m_iData = EFIM_GetOneDataID(
                      EFIM_GetRawDataTypeIDbyMySQLType("integer"),
```

```
                    "601"
                );
```

Because the translator target is producing sound SQL statements, it is also possible that a translator can generate an SQL statement without E-model SQL APIs. Query 10.11 shows an example translated into a complete SQL statement without APIs. Here you see tables and relations that are based on the current E-model database layout depicted in Figure 52. However, the output is complex and difficult to understand compared with Query 10.10. Designing a translator without fundamental APIs is very hard to implement. Thus, our current work pursues development of a translator that outputs an SQL statement like Query 10.10.

Query 10.11. Raw level converted SQL statements of Query 10.9.

```
SELECT
  srd.data,
  601,
  se.m_iUpdateLog
FROM
( SELECT
    e.m_iSymbol,
    601,
    e.m_iUpdateLog
  FROM
  200_event AS e
  INNER JOIN 100_data AS d ON (e.m_iData = d.m_iIndex)
  INNER JOIN 114_data_integer AS rd ON (d.m_iRawDataID = rd.m_iIndex)
  WHERE
    d.m_iRawDataType = (
      SELECT m_iIndex
      FROM 121_data_type_rawdata
      WHERE m_sDataName = "integer"
```

```
    ) AND
    rd.data = 601
) AS se
INNER JOIN 100_data AS sd ON (se.m_iSymbol = sd.m_iIndex)
INNER JOIN 116_data_varchar AS srd ON (sd.m_iRawDataID = srd.m_iIndex)
WHERE sd.m_iRawDataType = (
  SELECT m_iIndex
  FROM 121_data_type_rawdata
  WHERE m_sDataName = "varchar"
)
```

10.4.2 EML parsing mode

If the E-model system operates as a stand-alone as in Figure 65, all EML queries should be parsed into optimized SQL statements using E-model SQL APIs for the E-model system query. However, if a query is a structured one that exactly specifies the target database and its schema, and if the E-model system coexists with companion databases, then a translation process is not necessary and the system will forward the query to the target database.

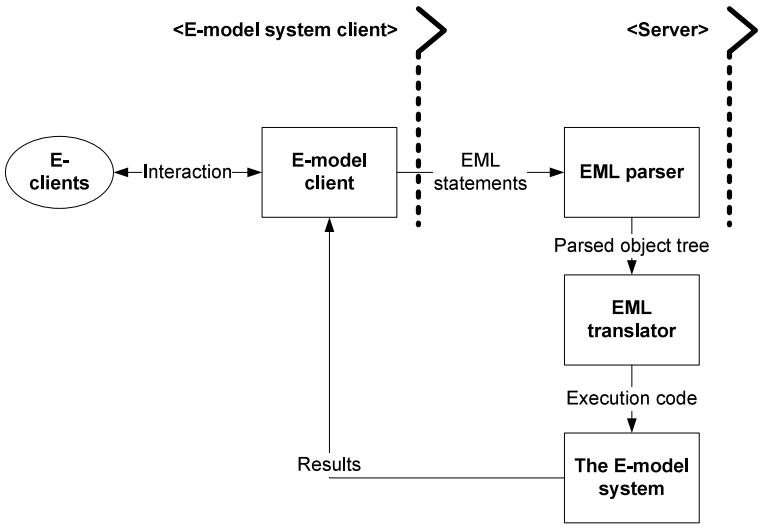


Figure 65: The stand-alone E-model system configuration.

Thus, in an MD-3 system configuration such as Figure 64, the EML parser works in

a mixed mode. Because EML is a SQL-89 compatible language, a user can change the running mode freely. If he or she wants to access the target database directly, the EML translator will do so. If a statement includes any EML namespace, then the EML translator will optimize the query, based on the records in the E-model system, to locate the instance at the target database. If the statement does not specify any target database (i.e., FROM _*), then the output will be directly calculated from the E-model system.

Table 13: The EML extended functions.

Name	Notation	Comments
Relation	_R(„)	Relation search query. (Q_r in Section 4.3)
Graph path	(%d, _EPATH, *)	_EPATH specifies edges to walk. %d notes the dth selected field.
Popularity	_Popularity()	Used to order results by popularity
Semantic expansion when using WordNet [24]		
Name	Notation	
Antonym	_ANTPTR()	
Similar	_SIMPTR()	
Synonym	_SYNS()	
See also	_SEEALSO()	
Attribute	_ATTRIBUTE()	
Verb group	_VERBGROUP()	
Participle	_PPLPTR()	
Pertainym	_PERTPTR()	
Derivation	_DERIVATION()	
Domain	_CLASSIFICATION()	
Member	_CLASS()	
Domain category	_CLASSIF_CATEGORY()	
Domain usage	_CLASSIF_USAGE()	
Domain region	_CLASSIF_USAGEDRN()	
Domain member category	_CLASS_CATEGORY()	
Domain member usage	_CLASS_USAGE()	
Domain member region	_CLASS_USAGEDRN()	
Hypernym	_HYPERPTR()	
Hyponym	_HYPOPTR()	
Instance of	_INSTANCE()	
Part holonym	_PHN()	
Part meronym	_HASPARTPTR()	
Member holonym	_HASMEMBERPTR()	
Member meronym	_ISMEMBERPTR()	
Substance holonym	_ISSTUFFPTR()	
Substance meronym	_ISSTUFFPTR()	

CHAPTER XI

PERFORMANCE AND USABILITY EVALUATION

This chapter evaluates two aspects of the current E-model prototype: (1) query performance and (2) usability for domain application development. The theoretical computational speed boundary of the E-model is compared in Section 11.1 with that of popular database models. For standardized evaluation, Section 11.2 reviews popular database benchmarks. And within those, TPC-H [35] is selected because it is an industrial database benchmarking tool developed for a relational database and the E-model prototype is developed on top of a relational database. Thus, search cost comparisons with relational databases can be performed under the same server configuration. TPC-H benchmark results reviewed in Section 11.3 clearly show those situations in which the E-model system is superior to others, even when the E-model system is in its MD-1 stand-alone server configuration. Section 11.4 reviews the results of the dynamic graph SBV speed of the E-model in various situations. In the dynamic graph SBV experiment, the E-model's unified storage structure and fully indexed data objects that are stored in a small number of tables show performance that is superior to popular data models by several orders of magnitude.

The current E-model prototype is a sort of hybrid database specialized for graph data structures in which time and disparate data model handling is important. However, its application areas are not restricted to limited applications because the E-model provides rich semantics and high-order relations to interoperate with existing data models (See Chapter 9). Thus, we have developed a demo application under MD-1 configuration in Section 11.5. Because the E-model system was born from a concept of events in which time is the most critical change factor, it can naturally support dynamic multimedia information processing. This demo shows hybrid searching on video content using low-level video object features,

space, time, and associated disparate data models. It is a Web application in the Apache-PHP-MySQL servers, which is the most popular server configuration for Web services. Section 11.5 also reports several case studies for system review in terms of implementation.

11.1 Search cost comparison by data models

A search cost to find data of interest from a database system depends on the nature and properties of the source data objects and the database's embodied data structure. By search cost, we mean query time and the expenses to operate and maintain the database system. The query time can be interpreted many ways; For structured data models with prepared statements, a query time is the duration of a statement transaction. For unstructured data models, a query could not specify internal relations, and thus, a user may need to interact with the system repeatedly to find objects of interest. In this latter scenario, statement transactions could be issued multiple times. This will result in a much longer query time than with structured queries. In fact, inducing a quantitative query evaluation model is cumbersome and heuristic. For a generalized cost evaluation, let us use several measures and terms to evaluate the complexity of search algorithms for data models.

This section provides details of such measures used in Table 14 that lists the theoretical limit of E-model search cost compared with popular data models. Table 14 compares the speed of queries of relational, XML, graph, and the E-model for three cases: (1) structured query, (2) dynamic query, and (3) dynamic query with weighting factors.

A structured query means that a query statement exactly specifies the structure of the data objects and the way they are related to each other. It is assumed that a user knows all definitions like database and table structures, field names, and their raw data types, and in the case of XML, the sibling order between elements. Structured queries thus can take advantage of existing industrial standards like SQL for relational databases and XPath for XML-enabled databases.

A dynamic query represents a sequence of queries starting from an unstructured query in a SBV form and then retrieving related data objects by the relation structure that each data model supports. In retrieving related data, we assume that a system automatically analyzes the metadata on linked relations. Thus, a dynamic query simulates a typical user's search behavior on a Web search engine.

Table 14: Search cost comparison.

	Structured query	Dynamic query (DQ)	DQ with weighting factors
RDBMS	SQL, $O(\|E\|)$	N/A. Additional metadata analysis is necessary to search all tables and their columns.	N/A. Assumes a uniform probability.
XML	XQuery, $O(\|E\|\|C\|)$	XPath only. Must visit all elements in all XML documents.	N/A. Assumes a uniform probability.
Graph search (Assuming one graph)	N/A	Breadth-first search: $O(b^d)$, Depth-first search: $O(b^d)$, Bidirectional search: $O(b^{d/2})$.	For informed (heuristic) search, application tailored weights should be developed.
E-model	Materialized views	M-algorithm (Section 7.7.2)	RDAG supports
	Identical performance to RDBMS, $O(\|E\|)$, and to XML-enabled relational databases, $O(\|E\|\|C\|)$.	For RDAG, M-algorithm performs like bidirectional search, $O(b^{d/2})$.	Transaction time window, confined categories, confined data names, raw data type limit and reference count ranking system $\ll O(b^{d/2})$
	Search over RDAG		
	C-data query and retrieval steps are necessary to locate e-nodes of interest: $\simeq O(\|E\|) + O(\|N\|)$		

Each data model in Table 14 has its own search algorithms in which the design philosophy could cause differences in search cost measurement. This led to adoption of Big-*O* notation[1] to represent a generalized order of algorithm complexity by the inclusion of important variables that can be commonly applied to data models. Big-*O* notation describes the limiting behavior of a function when the argument tends towards a particular value or

[1] http://en.wikipedia.org/wiki/Big-O_notation

infinity, usually in terms of simpler functions. Big-*O* notation allows its users to simplify functions in order to concentrate on their growth rates: Different functions with the same growth rate may be represented using the same O notation. Variables and functions used in Big-O notations are listed below :

- ||: The cardinality (count) of the set.
- *E*: The count of relational joins for related databases, or the number of involved schemas for XML, or the count of edges (relations) in RDAG.
- *C*: The number of relational tables or XML documents or e-category nodes in RDAG.
- *N*: Counts total table records or XML elements or e-nodes.
- *b*: The branching factor (number of children of each node).
- *d*: The depth limit in the tree structure.

Based on the above definitions, let us review the search cost of structured queries for each data model. A graph search is assumed to be an unstructured model and thus, is not included here.

Relational model The speed of a query using SQL depends linearly on the number of joins. A simple join for two tables must query each table and merge matching records that satisfy search constraints on values and relations. To join more tables, this process should be repeated for all joins. Hence, in SQL, query speed declines in proportion to the number of joins in a linear order, $O(|E|)$.

XML The XML query language standard, XQuery [19], does not support XInclude in its current specification in a way that permits a reference between different XML schemas. Hence XQuery is limited to one document at one time and we have to repeat or program to expand XQuery over all XML collections. Such complexity thus increases the cost of $O(|E||C|)$.

E-model The E-model system supports two query methods under the MD-1 stand-alone configuration for structured queries: (1) the materialized view (See Section 9.4) and (2) search-by-value and navigate through a RDAG. The materialized view will provide a speed equivalent to the speed of the data model that the E-model exported to. In the second case, the dynamic query is composed of four steps: (1) Find c-data sets that match the value for the query, (2) retrieve the set of e-nodes that refers to the c-data sets, and (3) navigate through a RDAG by the structure of an associated e-category to find related e-nodes, and (4) finally return the trace of the e-nodes as the result. As for the c-data search as the first step, it is constant because all data objects are fully indexed. Thus, the search cost depends only on the order of relations and data counts: $\simeq O(|E|) + O(|N|)$.

As for dynamic queries, neither SQL nor XQuery supports a SBV query. Thus, we need some additional work for relational databases and XML to check all records over all tables or documents. For relational databases, a user must infer INFORMATION_SCHEMA to get information on all tables and databases to search. For XML, all XML schema documents in the database should first be analyzed to parse all XML documents to find a value of interest. The search cost of both approaches varies by the number of tables and documents. Formal definitions will be derived in Section 11.4.

Graph A typical graph search algorithm represents the order of complexity by using the number of children, b, and the depth, d, of each node. In this context, let us review three famous graph search models. Breadth-first search (BFS) [82] is a graph search algorithm that begins at the root node and explores all the neighboring nodes. Then for each of those nearest nodes, it explores their unexplored neighbor nodes and so on until it reaches its goal. In contrast, depth-first search (DFS) [82] is an algorithm for traversing or searching a tree, tree structure, or graph. One starts at the root (selecting some node as the root in the case of a graph) and explores as far as possible along each branch before backtracking. The theoretical boundary of the computing cost of a breadth-first search and a depth-first search

is equivalent to $O(b^d)$ [82], assuming the number of vertices and edges in the graph are not known ahead of time. Bidirectional search (BDS) [125] is a graph search algorithm that in a directed graph finds the shortest path from an initial vertex to a goal vertex. Because BDS runs two simultaneous searches (one forward from the initial state and one backward from the goal) and stops when the two meet in the middle, its cost is $O(b^{d/2})$.

E-model Because a RDAG of the E-model is also a graph that can be parsed into the composition of trees (See Section 9.2.3 for proof), the above graph-searching algorithms can be applied to a RDAG. The M-algorithm introduced in Section 7.7.2 is a bidirectional search [124, 37] from the viewpoint that we start from both ends to find the matching e-group nodes to connect source and target e-nodes.

Many approaches have been developed to enhance graph search speed. They are mostly classified as informed (or heuristic) searches [117, 83]. In an informed search, a heuristic that is specific to the problem is used as a guide. A good heuristic will make an informed search dramatically outperform any uninformed search. Table 14 classifies such heuristics for dynamic queries with weighting factors. It should be noted that the E-model can support rich weighing factors in this context like a transaction time window, confined categories, data names, raw-data type limits, and reference counts that significantly reduce computational complexity: $\ll O(b^{d/2})$.

11.2 Reviews of database benchmarks for structured query

Typical database benchmarks are concerned with the query time or the ratio of the query time to the system cost. This section will review benchmarks roughly for three data models:

Relational databases TPC-H [35] was devised for relational database systems. It is a decision support benchmark that consists of a suite of business-oriented ad-hoc queries and concurrent data modifications. The queries and the data populating the database have been chosen to have broad industrywide relevance while maintaining a sufficient degree of ease

of implementation. This benchmark illustrates decision support systems that examine large volumes of data, execute queries with a high degree of complexity, and finally give answers to critical business questions. TPC-H evaluates the performance of various decision support systems by execution of sets of queries against a standard database under controlled conditions.

XML As for hierarchical data models, XMark [120], XMach-1 [21] and XOO7 [95] have been proposed as specialized for XML documents. They are designed to help both implementers and users compare XML databases in a standardized application scenario. Benchmarks are composed of a set of queries in which each query is intended to challenge a particular aspect of the query processor. These benchmarks provide predefined ad-hoc queries of the structured data models and have cost models for comparison.

Text information retrieval For unstructured query evaluation, Text REtrieval conference (TREC) [29, 134] and its Million Query (1MQ) Track [4] has been a popular measurements for text information retrieval. 1MQ is an exploration of ad-hoc retrieval on a large collection of documents. It investigates questions of system evaluation, particularly whether it is better to evaluate using many shallow judgments or fewer, but thorough, judgments. Participants run 10,000 queries against a 426Gb collection of documents at least once and judge documents for relevance with respect to some number of queries.

Because the current E-model prototype is developed on top of a relational database, we selected TPC-H for evaluation of structured query performance. However, database system evaluation involves consideration of other factors. When large amounts of data are involved, three factors affect query performance: (1) data models and their relations, (2) database architecture, and (3) complexity of the query. As for database architecture, relational databases are optimized for rectilinear table types and their joins, whereas XML is convenient to express tree-structured information sets. This means that there are tradeoffs

between query performance and design freedom. Thus, in the following sections we evaluate the E-model system in two ways: (1) structured query performance using TPC-H and (2) dynamic query performance using search-by-value experiments.

11.3 Structured query performance review: TPC-H

As we discussed in Section 11.1, the search cost for the E-model is more than for that of a relational database (See Table 14): $O(|E|) + O(|N|)) >= O(|E|)$. In terms of theoretical boundaries, the E-model performs a structured query slower than a relational database. However, this situation is reversed with a dynamic query (See Section 11.4).

Thus, this section uses the TPC-H query set to evaluate quantitatively how much slower the E-model system is than a relational database. The result reported in this section is meaningful for a system designer who needs to decide if the E-model system should run under the MD-1 stand-alone configuration without materialized views. This situation could arise if a system needs frequent insertion and requires too many materialized views to maintain in real time with the available computing resources. Thus, faced with such a decision, it is important to know which types of structured queries the E-model system handles well or poorly.

11.3.1 Experiment environment

TPC-H[2] provides standardized tools to generate test data, table definitions to setup the database, and a set of queries to evaluate performance. Actual query time varies significantly according to the amount of data to test. However, comparative results between database systems are not that affected by data size. For our experiment, we set up identical operating environments for both the relational database (MySQL) and the E-model system. TPC-H suggests 22 types of queries in which a user may change some values. We chose the default value that TPC-H recommends. Figure 66 is excerpted from the TPC-H manual

[2] http://www.tpc.org/tpch/

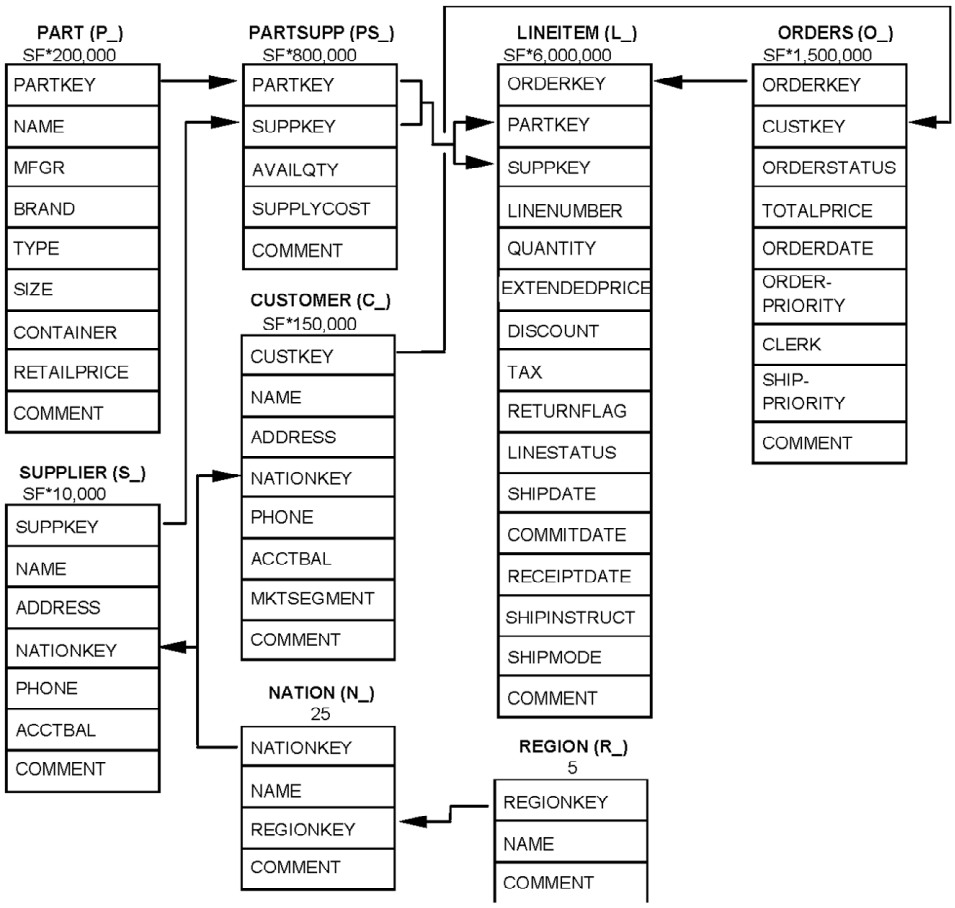


Figure 66: TPC-H database schema (Excerpted from TPC-H manual 2.7.0, Figure2); Source: http://www.tpc.org/tpch/spec/tpch2.7.0.pdf.

to show the database layout composed of eight tables in which all TPC-H queries are prepared. Within 22 queries, we selected two (numbers 5 and 6) for evaluation because Query 5 needs six tables to join in computation, whereas TPC-H's Query 6 needs only one table for query computation.

Query 11.1. TPC-H Query 5 lists the revenue volume done through local suppliers.

```
select
        nation.name,
        sum(lineitem.extendedprice * (1 - lineitem.discount)) as revenue
from
        customer,
        orders,
```

```
        lineitem,
        supplier,
        nation,
        region
where
        customer.custkey = orders.custkey
        and lineitem.orderkey = orders.orderkey
        and lineitem.suppkey = supplier.suppkey
        and customer.nationkey = supplier.nationkey
        and supplier.nationkey = nation.nationkey
        and nation.regionkey = region.regionkey
        and region.name = 'ASIA'
        and orders.orderdate >= date '1994-01-01'
        and orders.orderdate < date '1994-01-01' + interval '1' year
group by
        nation.name
order by
        revenue desc;
```

Query 11.2. TPC-H Query 6 quantifies the amount of revenue increase that would have resulted from eliminating certain companywide discounts in a given percentage range in a given year:

```
select
        sum(extendedprice * discount) as revenue
from
        lineitem
where
        shipdate >= date '1994-01-01'
        and shipdate < date '1994-01-01' + interval '1' year
        and discount between .06 - 0.01 and .06 + 0.01
        and quantity < 24;
```

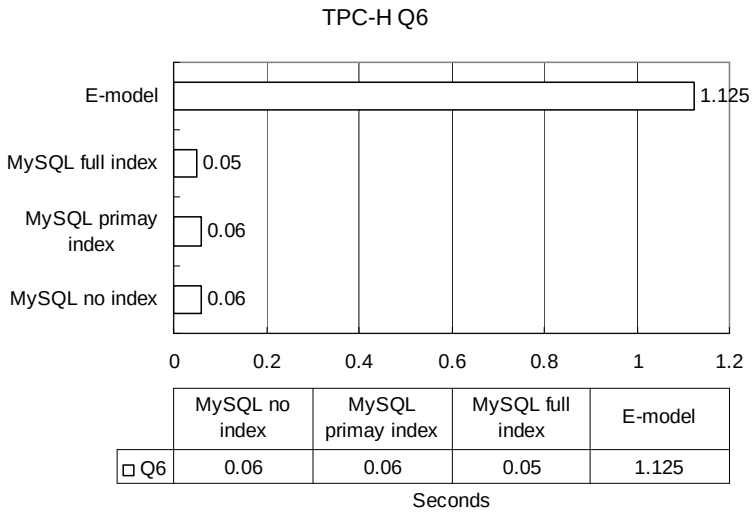


Figure 67: TPC-H query 6 result in mixed indexing modes.

11.3.2 Results

Let us see the TPC-H Query 6 result first depicted in Figure 67 that handles only one table. As expected by the computing cost in Table 14, the E-model is much slower than the direct query issued to MySQL. This is because the original query opens only one table, *lineitem*, which does not needs joins with any other table index. Also, the records (67,015) and indexes that must be copied and handled are much fewer than that of the E-model.

As for MySQL , we tested indexing methods in three modes: (1) *MySQL no index* means no index on fields, (2) *MySQL primary index* has only the primary index specified in the TPC-H table definition, and (3) *MySQL full index* has full indexes for all fields in a table. In the case of Query 6, different indexing methods do not affect query performance that much because it only handles a few fields in simple computation.

However, on Query 5, the E-model outperforms the *MySQL no index* method as depicted in Figure 68. Moreover, the difference in query speeds with other indexing methods gets much smaller than that with Query 6. Let us first see the results from the E-model and MySQL indexing methods. As mentioned before, the query speed in relational databases slows, depending on the number of joins. The E-model also needs to walk through the

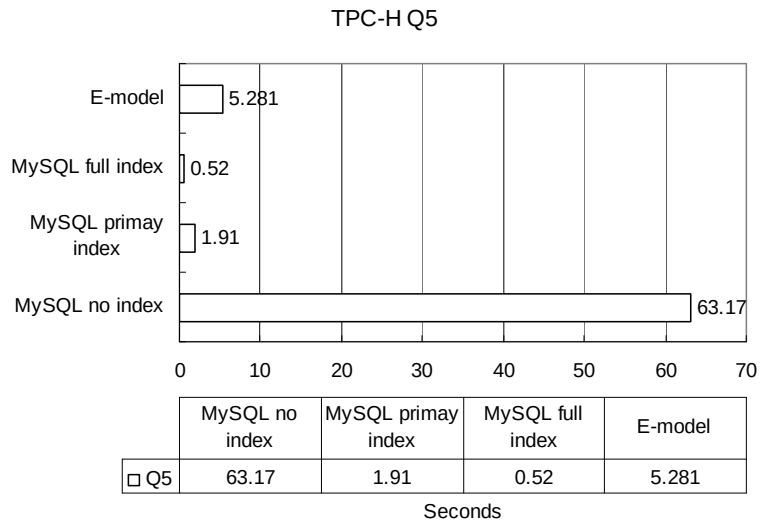


Figure 68: TPC-H query 5 result in mixed indexing modes.

related e-nodes in a similar manner. However, the E-model looks only for connected e-nodes necessary for joins, whereas to perform table joins, the relational database query first searches all fields in the statement and then compares the foreign key. Although the E-model still has overhead in c-data object processing that makes it slower than the indexed MySQL cases, the performance gap gets much smaller than in the case of Query 6.

In the case of the *MySQL* method with no index of fields, this violates the theoretical boundary: $O(|E|) + O(|N|)) \geq O(|E|)$. It should be noted that in calculating the computing cost of a relational database search, we assumed that the field of interests are indexed. Thus the search of the field of one table is assumed to be constant. However, *MySQL with no index* method does not have any index. In other words, all fields necessary for table joins do not have an index to compare values between tables. Thus a query should look for all values in the records to find matches to join. And this happens to all tables involved in the join. So for *MySQL with no index* method, the computing cost in Big-*O* notation becomes $O(|E||C|)$, which is much bigger than that of the E-model: $O(|E|) + O(|N|)) \ll O(|E||C|)$.

MySQL with no index simulates the situation with complex data models in which tables and their relations are not properly indexed. This happens with applications in dynamic environments in which new data models are frequently inserted into the existing model.

Stock management is a situation in which a variety of disparate data sources are introduced and disappeared daily. This problem is a well-known chronic challenge in database management and also where the E-model can make a good contribution.

11.4 Dynamic query: Search-by-value performance review

SBV is an information retrieval query model, especially for text document search engines. The simple input window of most Web search engines is an exact SBV interface that encapsulates their complex internal data models while providing a single interface for all queries about their content. This section compares the SBV performance of the E-model system with popular storage engines to test the power of the E-model for complex data models. In this experiment, our test set is not limited to tokenized words but involves many structured data objects in complex relations.

When disparate data models are involved in SBV, it needs to identify the schema of an associated value to return structured results for presentation. In a relational model, such schema metadata are grouped in a set of three tuples: a column, a table, and a database. In an XML model, it would be an element with its path from the root, a XML document, and a XML collection.

SBV is useful for a user who wants to first identify in which data model an associated value is involved. On the other hand, a user who has a structured query needs to compose the query statement with an exact object value with its exact storage objects in which the value should exist. SQL is an example in which a user must specify exactly which database, table, and column name to find and join tables. XPath [28], which is a query language for XML, also needs a correct path string from the document root to locate the object value. Therefore, to perform SBV for structured databases, we first need to look through the metadata on database objects like: (1) SQL/schemata, which is an extension of the SQL standard [43] and includes the definitions of databases for a relational database; and (2) XML schema for XML documents.

11.4.1 Experiment environments

As the source for SBV experiments, we selected a video metadata search application as an example of a complex query. The set of data objects used in the experiment is composed of 14 schemas of sitcom-related information as specified in Table 15. Each schema and its relations with others is depicted in Figure 69 as an entity-relationship diagram.

Table 15: SBV experiment specification.

Object	Find e-nodes in E-model (records for RDBMS or elements for XML) from an arbitrary value.
Data	NBC *Friends* sitcom season 6, 25 episodes. 14 data schemas: Episode, script, scene, scene lookup table, scene similarity, scene sentence, subtitle, subtitle sentence, scene similarity, scene sentence similarity, word links, word, video face images, video thumbnail. Total 98 columns (elements for XML).
Method	
RDBMS	Use SQL/Schemata to find tables and columns in which the SBV value exists.
XML	From given 14 XML schemas, use XPath to query the SBV value for all documents. Returns the complete path of the found elements.
E-model	Find all e-nodes with the SBV value.

In this experiment, three database systems will be compared with each other: (1) a relational database, (2) XML, and (3) the E-model. Based on the database layout in Figure 69, we created 14 XML schema to generate XML documents. For the E-model, we assumed the MD-1 stand-alone database configuration and all data objects in 14 tables were imported into the E-model system (See Section 9.1.2 for E-model-to-relational interoperation algorithms).

To have all three data models run under the same server configuration, we chose the XML-enabled relational database method, which stores each single XML document into one record and then uses XPath to query the value. Thus, one relational table to store XML documents has two columns: (1) a primary ID column for identification and (2) a TEXT column to store an XML document. We assumed that all XML documents in one table conform to one XML schema. Thus, 14 tables were prepared to store XML documents for

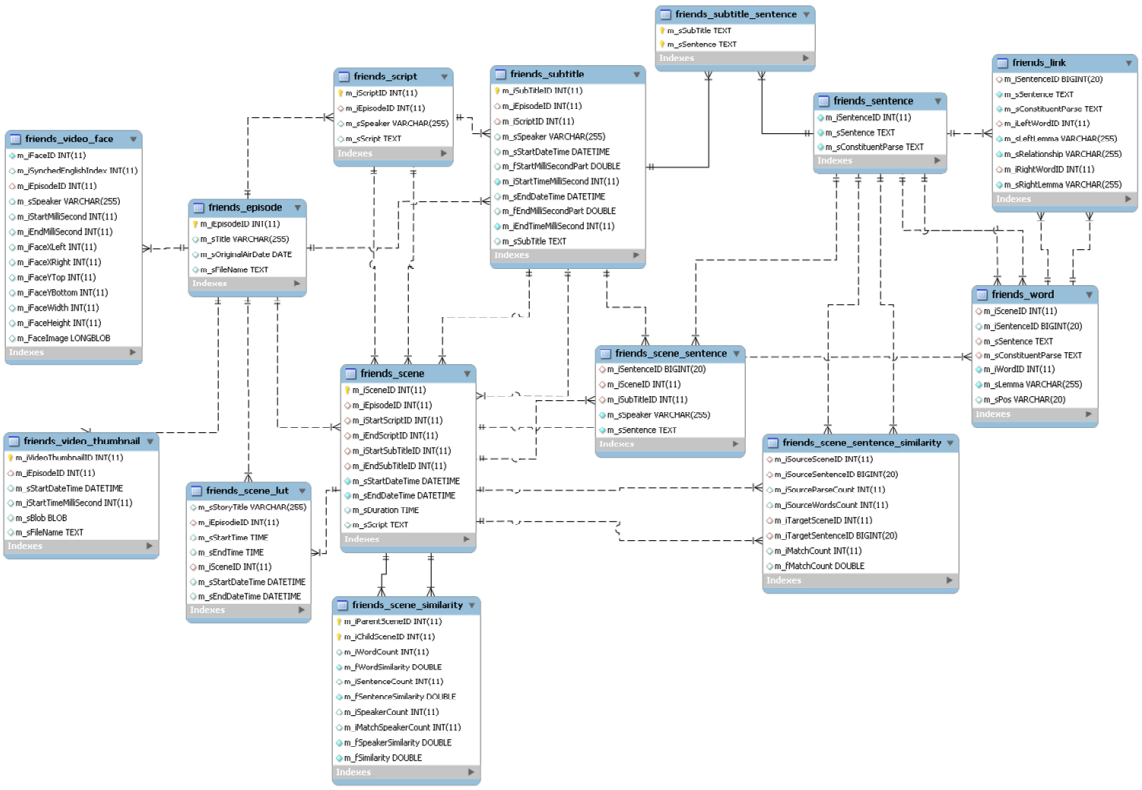


Figure 69: SBV data NBC *Friends* sitcom schema.

each XML schema.

The method to perform SBV for the E-model has already been introduced in Section 4.2, which dealt with the e-node search constraints and search functions to query. Methods for the other two data models are summarized below.

Relational model Based on the SQL/schemata standard introduced earlier, schema definitions were stored in the INFORMATION_SCHEMA view. Therefore, we retrieved all table definitions and their fields. Second, a given value was compared with each column. In this configuration, we may expect that the SBV performance depends on the number of total columns of all tables within a database to compare.

XML Regarding XML cases, XPath [15] is used to retrieve the value of each element. To use XPath, a path to an element should be known a priori that is specified in an associated

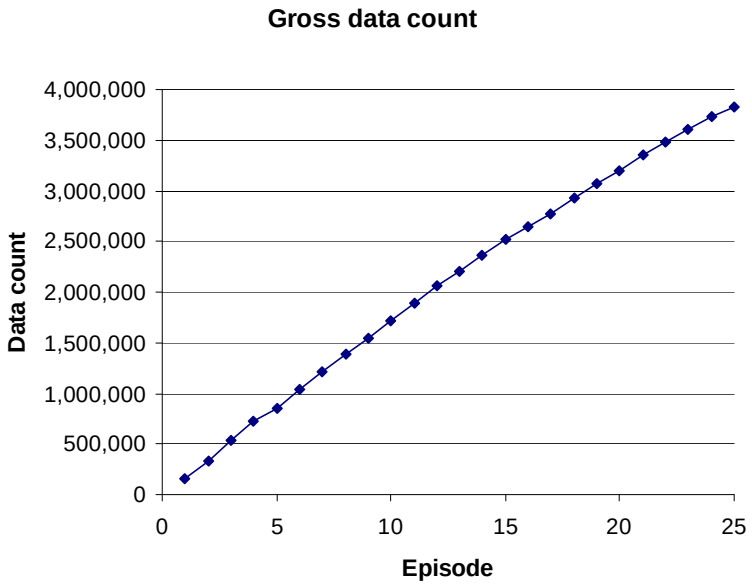


Figure 70: Episodes data gross count.

XML Schema. The sibling order for each element should be specified so as to retrieve adjacent elements completely. For instance, "/Product/Title[3]/price[2]" in an XPath expression means the second "price" value from under the "Product" root node and its third descendant "Title" parent. Thus, for each element node we first used the count XPath function (ex. count("Product/Title[3]/count(price)")) to retrieve the total count of elements, then retrieved a value from each element iteratively to compare each element with a given SBV value.

11.4.2 Experiment result review

In the experiment, we performed a query for selected episodes that incrementally adds one episode at every query. This was to check performance versus the data size of the query. Figure 70 shows the gross data count for 25 episodes. For instance with a relational database, it is the total count of tables cells within selected episodes. For all 25 episodes, 3.8 million data values were queried in this experiment.

As a result, the E-model system showed superior performance in the SBV compared with the relational and XML models (17 times faster than the relational database and 256

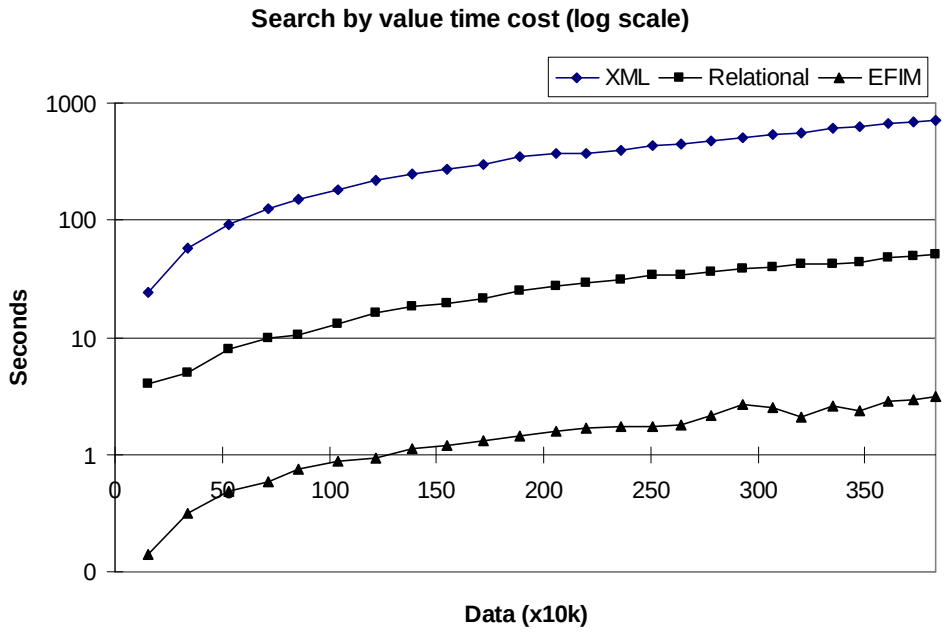


Figure 71: Search-by-value time cost comparison in the log scale.

times faster than the XML-enabled database). A search speed comparison chart for all episodes is depicted in Figure 71. The data count in the X-axis is the count of all records in all tables in selected episodes. It should be noted that this graph is drawn in the logy scale. As it clearly shows, search speed is compared in order of magnitude. Differences in the SBV query for the full 25 episodes are highlighted in Figure 72. The number that appears at each bar is a relative comparison assuming the speed of XML as 1.

11.5 Case study: e-Friends for multimedia video application

NBC's *Friends* sitcom video database is actually used for application implementation on top of the stand-alone E-model system. The SBV method introduced in Section 11.4 searches seed e-nodes to expand the query over related e-nodes. In doing so, various constraints over rich relations (See Chapter 4) can be applied for efficient and complex queries. This section introduces several case studies that e-Friends, which is the name of our first demo E-model application, actually implemented.

The purpose of this demo application is to demonstrate the extensive query capability

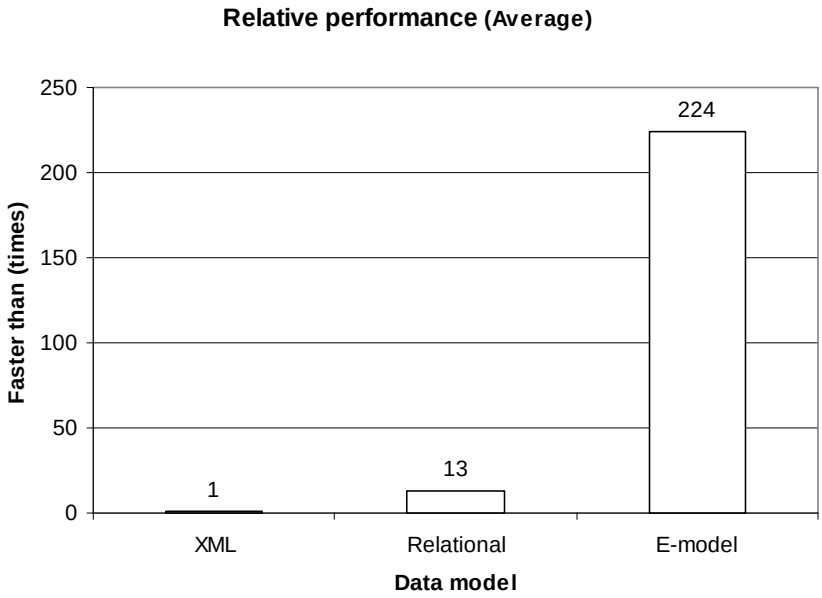


Figure 72: Relative SBV search speed comparison (Relative to the XML query as 1).

of the E-model system. Search interfaces shown in Figure 73 were focused on finding who, when, what, and where queries that we will handle in the following examples. Each example uses the new E-model language, EML, proposed in Chapter 10.

11.5.1 Example 1: Find script sentences of interest

In searching script sentences that include a specific word of interest, all we have is an assumption that it is "sentence" that includes a composite set of keywords: "marry | married | marriage | annulment." This is similar to Query 10.3 that uses EML namespace elements as follows:

Query 11.3. Search subtitle sentences.

```
SELECT *
FROM _* AS t1
WHERE
  t1._name LIKE "subtitle" AND
  t1._value REGEXP "marry|married|marriage|annulment";
```

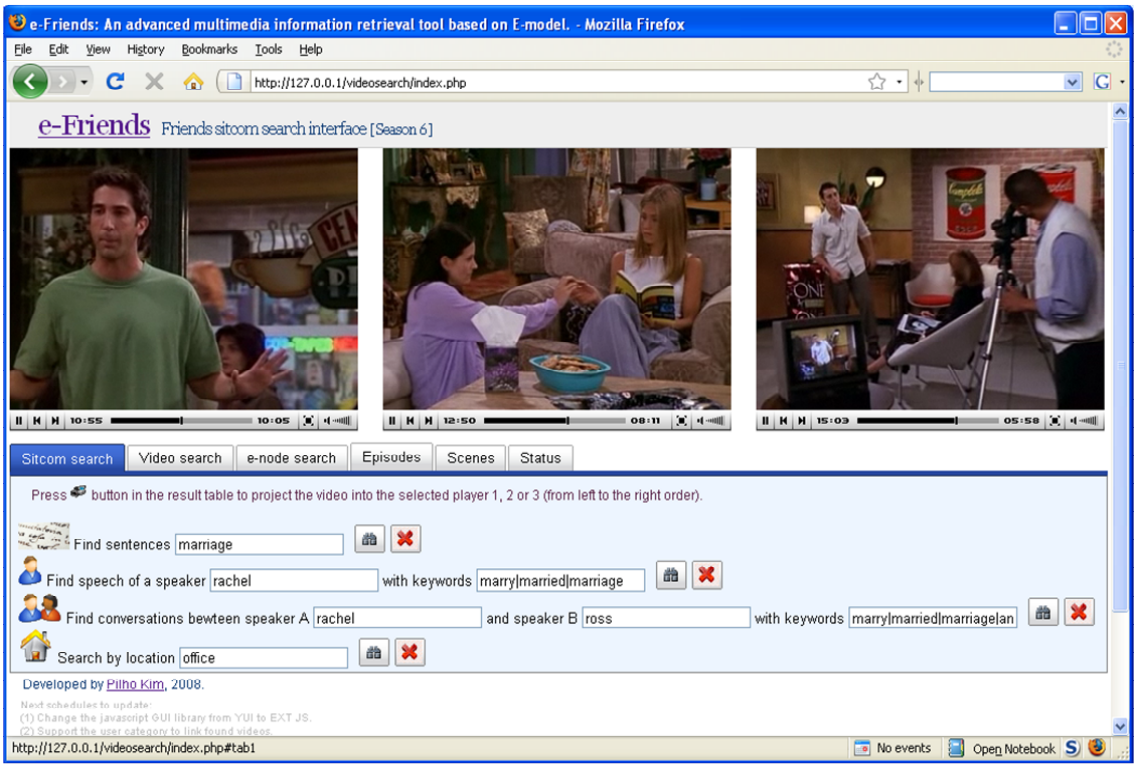


Figure 73: e-Friends interface for sitcom multimedia search application.

It should be noted that the EML statement in Query 11.3 permits an uncertain naming convention LIKE “subtitle” in the field name. This is impossible in SQL because the matching field name in the relational database is the ”m_sSubTitle” column in the ”friends_subtitle” table and must be specified exactly in the query statement.

11.5.2 Example 2: Find who said that and when

Let us extend Example 1 to find who spoke a certain word and when. Because the E-model naturally supports transaction times, we assume that all e-nodes are temporally synchronized with the real-world time of an event. Based on the database layout in Figure 69, a column named like “speaker” does not exist in the “subtitle” table because in practice subtitles embedded in DVD media include only the timestamp and subtitle text. Such speaker

information is found in the script. Thus, this query should look for two tables (two e-categories for the E-model) and find matches by values. This is the table join in a relational database. Query 11.4 solves the join problem simply by matching the transaction timestamps of e-group nodes ("t1._parent" and "t2._parent"). Unlike in SQL, this approach does not need any additional value comparison between the two tables.

Query 11.4. Find a speaker and time of speech.

```
SELECT t1.*, t1._timestamp, t2.*, t2._timestamp
FROM
  _* AS t1,
  _* AS t2
WHERE
  ( t1._name LIKE "subtitle" AND
    t1._value REGEXP "marry|married|marriage|annulment") AND
  ((t2._name LIKE "speaker") OR
   (t2._name REGEXP "start" AND t2._type = "DATETIME"))
WHEN t1._parent = t2._parent;
```

11.5.3 Example 3: Search a conversation between two speakers on some topic

This case is a little complex because we first must define *Conversation*. What we want to find is a conversation between two speakers that includes specific keywords. This kind of high-level query requires composition of the query in a sequence that reflects knowledge of the domain application. A conversation means that two or more speakers talked to each other. Their discussion topic should coincide, and the temporal gap between the two speeches should be less than some reasonable time limit.

This query should be done in a bidirectional way, which means that if one speaker says a keyword of interest, then we expand the query result around that speech event to find other speeches of interest. Such expansion will be bounded within the same episode, same scene, and within a time limit (for instance, a 30-second limit before or after the event).

Because this will work bidirectionally in temporal order, a set of speeches could be found if both speakers mentioned a keyword of interest during the conversation.

Let us compose this high-level query in both EML and SQL for comparison.

Query 11.5. Search conversation in EML.

```
SELECT t3.*, t4.*
FROM
( SELECT t1._parent
  FROM _* AS t1,
       _* AS t2
  WHERE (t2._name LIKE "speaker" AND t2._value LIKE "rachel") AND
        (t1._name LIKE "subtitle" AND
         t1._value REGEXP "marry|married|marriage|annulment")
  WHEN t1._parent = t2._parent
) AS t3,
( SELECT t1._parent
  FROM _* AS t1,
       _* AS t2
  WHERE (t2._name LIKE "speaker" AND t2._value LIKE "ross") AND
        (t1._name LIKE "subtitle" AND
         t1._value REGEXP "marry|married|marriage|annulment")
  WHEN t1._parent = t2._parent
) AS t4
WHEN ABS(t3- t4) <= 30;
```

Query 11.6. Search conversation in SQL.

```
SELECT t3.*, t4.*
FROM
(
SELECT t1.m_iEpisodeID, t1.m_iSubTitleID, t1.m_sStartDateTime,
       t2.m_iScriptID, t2.m_sSpeaker, t1.m_sSubTitle
FROM
friends_subtitle AS t1
```

```
INNER JOIN friends_script AS t2 ON (t1.m_iScriptID = t2.m_iScriptID)
WHERE t2.m_sSpeaker LIKE "rachel" AND
      t1.m_sSubTitle REGEXP "marry|married|marriage|annulment" AND
      t1.m_iEpisodeID = t2.m_iEpisodeID
) AS t3,
(
SELECT t1.m_iEpisodeID, t1.m_iSubTitleID, t1.m_sStartDateTime,
       t2.m_iScriptID, t2.m_sSpeaker, t1.m_sSubTitle
FROM
friends_subtitle AS t1
INNER JOIN friends_script AS t2 ON (t1.m_iScriptID = t2.m_iScriptID)
WHERE t2.m_sSpeaker LIKE "ross" AND
      t1.m_sSubTitle REGEXP "marry|married|marriage|annulment" AND
      t1.m_iEpisodeID = t2.m_iEpisodeID
) AS t4
WHERE
    ABS(t3.m_sStartDateTime - t4.m_sStartDateTime < 30) AND
    t3.m_iEpisodeID = t4.m_iEpisodeID;
```

Query 11.5 first selects two e-groups of speech event as in Query 11.4, then simply compares the transaction time gap between e-group nodes. For a comparison with a structured query, Query 11.6 is the actually working SQL statement for a “friends” relational database that extracts conversation events. The difference between the two queries displays clearly the difference between the two languages as summarized below:

ER-1 EML permits imprecise field name constraints, whereas SQL has to specify it exactly: (ex. *t2._name LIKE “speaker”* vs. *t2.m_sSpeaker*).

ER-2 Joins between two schema can be made by transaction time, whereas SQL needs to perform a value comparison that requires one additional table scan operation to look up the other field (*m_iEpisodeID*) to make a match: (ex. WHEN t1._parent = t2._parent vs. t1.m_iEpisodeID = t2.m_iEpisodeID).

The two features of EML mentioned above distinguish the usability of the E-model system for multimedia applications because convenience in writing a complex query for disparate multimedia information databases is a critical aspect in system implementation and maintenance. Examples introduced in this section are rather close to structured queries for comparison with structured databases. Users may get more benefits in writing complex queries or in system implementation with various constraints and functions listed in Chapter 4.

CHAPTER XII

CONCLUSION

This work demonstrates the steps involved in proprietary data model implementation. The E-model, which is the core product of this work, shows how data models progress to bridge the gap between disparate data models. We wish the E-model could help reduce impedance mismatches between multimedia and databases. Topics handled in this work start with fundamental data object design and end with computer language implementation. We hope this work will serve as a guide for implementation for a reader who has new ideas on how data objects should be formulated, how groups are created, and how they propagate to other data objects. This chapter concludes this work with reviews of the E-model.

12.1 Reviews of E-model

The E-model has many new ideas from its birth. Its design differs significantly from existing data models. Even introducing one small data object, the e-node introduced in Chapter 3, changes dramatically the way that conventional data models have insisted on operating. An e-node is unlike the cells in a rectilinear relational table; nor is it like any elements in XML. The e-node represents itself by a symbol and a data value, and both objects are again abstract and domain independent c-data objects. This unique architecture allows one c-data object to be used as either a symbol or as a data value. It is even possible that a c-data object can represent itself when no other c-object can represent it.

The c-data object proposed in Section 3.5 is an encapsulation of disparate raw data silos. Existing databases confine storage to a predefined form like tables and databases in relational databases or like documents and collections in XML. In the E-model system, the way to access raw data goes through the encapsulated c-data object interface because our design originates from our experiences with problems we encountered in handling disparate

raw data objects for multimedia archives. Moreover, different specialized indexing methods devised for each type of raw data interferes unified access to data objects. For instance, some applications may need a very fast indexing method for temporal objects, but others may need a method to index huge spatial objects. Consequently, c-data objects, which are a list of integer-type identity numbers in the E-model system, are devised to separate the data model from enacted raw data type-specific storages. However, this does not mean slower access to raw data values than with structured data models. Our experiment demonstrates that in comparison with complex data models, the E-model shows an order of superior performance with a SBV query.

The E-model permits rich types of relations between e-nodes. Furthermore, unlike any conventional data models, a relation is materialized as an object in Section 3.4 just like with any other objects that it connects to. This allows existing object query methods, their full constraints and functions for query can be equally applied to relations. Thus, when we look for information linked in special relations, we do not even need to navigate to a graph; instead we can query this special relation to selectively retrieve linked objects. This is like looking for graph edges of a specific color of interest. The three-tuple adjacency list devised in Section 7.6.1 as an E-model storage structure can perform this query in one transaction cycle.

As demonstrated in Chapter 11, many of the constraints developed in Chapter 4 for E-model objects significantly enhance the E-model query performance. Such actions as search on relations, semantic query expansion, inherited object temporal property, ranking orders, and measurement of eventfulness are well-known challenges that are not feasible to implement with conventional data models.

Finally, a new language, EML, for the E-model was introduced in Chapter 10. EML is the culmination of all efforts to provide a user the most convenient method to interact with the E-model system. With EML, a user does not need to know the details of the E-model system or its internal data structure. EML is designed to extend SQL, which is the most

popular database language. EML's presentation power and feasibility for complex query preparation is well documented in Chapter 10 and Chapter 11.

12.2 Contributions

When we first published a paper on an experiential meeting system [71] in 2003, its purpose was to allow people to be telepresent in a remote meeting and still be able to review proceedings of the meeting or of several meetings with full use of all the data recorded in a meeting. The data includes video, audio, presentations, text material, databases and Web sites related to people, the discussions in the meeting, and any other data or information that could be obtained related to the events in the meeting. We considered this meeting and full data access as a problem in management and of experiential access to all multimedia data acquired in a meeting.

For experiential access to live and archived meetings, we proposed detecting and storing events at three levels, domain, elemental, and data, all of which are similar to the E-model's objects, e-category, e-node, and c-data. We addressed in the earlier paper issues in organizing information at the domain level and in using current signal processing algorithms to detect events at the data level.

At this point, we started to devise primitive event models in pursuit of unifying disparate data models under the umbrella of events. The rudiments of such a conceptual model were proposed in 2004 [123], based on the notion of events in which the definition of an event and its fundamental properties were beginning to be settled upon. Specifically, the temporal aspects of an event were considered its most important aspects in defining a data object and its relations with other objects. In [56], we explored the area of continuous multimedia streaming in which continuous queries of multimedia events and their properties are most important. Our early proposed architecture for an event processing system for heterogeneous data management was published at [76]. It emphasized the importance of a user in the creation of semantically meaningful data. At this point, WordNet was first introduced

into our system design to extend semantic queries and to label data objects in a manner that a community of users could recognize without semantic confusion. This approach is implemented in the E-model as *Tag* in Section 8.5.

Since then our work has begun to exploit application areas that are natural fits with the concept of events. Personal chronicling tools (PCT) published in 2005 [79] were developed when this author was working in IBM research. The purpose of PCT was to help individuals to far more effectively retrieve and review their activities and interac-tions; at an enterprise level, the tools can be data mined to identify groups of common and complementary interests and skills or to identify implicit work processes that are commonplace in every enterprise. At that time, the existing tools for personal information management (PIM) were still rudimentary and limited in their support for dynamic capture of ongoing activities, in the organization and presentation of captured information, and in supporting rich annotation, search, retrieval, and publication of this information. PCT featured four primary tools: (1) event monitoring, (2) interactive annotation, (3) browse/search, and (4) edit/publish. Although PCT was mostly focused on tools for personal information chronicling in the enterprise environment, a new term *eChronicle* that we devised to symbolize our efforts on heterogeneous data management became our main research issue. In a similar spirit, our collaborators have started projects in pervasive computing (PICASSO) [58] in army operations (DARPA EC-ASSIST) [135, 112] and in various information chronicling projects [135, 112, 111, 89, 135].

An event-based multimedia chronicling system published in 2005 [75] began to define a generalized event as a materialized data object with formal definitions that are feasible for database processing. [75] was the first work to separate symbols and data values, then put events between them to make a distinct connection. Early concepts of c-data (See Section 3.4.3), composite events (See Section 3.4.2), and orderly linked events (See Chapter 7) were born from this paper. This was followed by the category-based functional information

model developed in 2006 [77], which featured the first use of an e-node concept to represent the identity of information and categorized relations to represent relations between data objects that became the e-functions of the E-model (See Section 5.2). That paper focused on the functional representation of data relations in which the flow of information over a network is the directed path of a graph and is an ancestor of the RDAG introduced in Chapter 7.

Since then we have poured significant efforts into embodying a concept of an event as a generic database system. The first E-model prototype implemented on top of a relational database began operation in late 2007. It was composed of hundreds of server-side stored procedures and functions that proved the E-model system can be implemented in any relational database. To enhance its performance and evaluate its algorithms, this prototype was tested against various disparate data objects, including the English Wikipedia database, personal information databases stored in XML collections, DVD movie scripts, and subtitle data objects. Some of this was to test temporal correlations between objects and to find an opportunity to enhance multimedia object pattern detection as well as improve segmentation results from spatio-temporal relations. All of them were merged in the final system test, and the E-model system performed well in the experiment. Based on this prototype, we developed the first demo application, *e-Friends*, in 2008, which provided an extensive search environment for video content retrieval (See Section 11.5). Currently, a paper on our new E-model system is under preparation for publication to [78].

In addition, this author had an opportunity to develop a traffic asset multimedia management system that requires processing of terabytes of outdoor street and road sign images. To maintain their traffic asset inventory databases, agencies have been manually reviewing millions of roadway video log images. This author proposed an innovative traffic sign image processing method that automatically detects traffic signs and their types and reduces by 80 percent the manual review requirements of the sign inventory workload. This work was published to [132].

APPENDIX A

E-MODEL CATEGORY XML SCHEMA

The complete E-model category XML schema is listed at Figure 74 and Figure 75.

```
<?xml version="1.0" encoding="UTF-8"?>
  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified" attributeFormDefault="unqualified">
    <xs:element name="e_category">
          <xs:annotation>
            <xs:documentation>Parent category</xs:documentation>
          </xs:annotation>
          <xs:complexType>
            <xs:sequence>
       <xs:element name="e_name" type="xs:string"/>
       <xs:element ref="e_sdtype" minOccurs="0" maxOccurs="unbounded"/>
       <xs:element ref="e_function" minOccurs="0" maxOccurs="unbounded"/>
       <xs:element ref="e_category" minOccurs="0" maxOccurs="unbounded">
         <xs:annotation>
               <xs:documentation>Child category</xs:documentation>
         </xs:annotation>
       </xs:element>
            </xs:sequence>
          </xs:complexType>
    </xs:element>
    <xs:element name="e_sdtype">
          <xs:complexType>
            <xs:sequence>
       <xs:element name="e_name" type="xs:string"/>
       <xs:element name="e_rawdatatype">
         <xs:simpleType>
               <xs:restriction base="xs:string">
                 <xs:enumeration value="blob"/>
                 <xs:enumeration value="datetime"/>
                 <xs:enumeration value="double"/>
                 <xs:enumeration value="integer"/>
                 <xs:enumeration value="text"/>
                 <xs:enumeration value="varchar"/>
                 <xs:enumeration value="geometry"/>
               </xs:restriction>
         </xs:simpleType>
```

Figure 74: The E-category XML schema, page 1.

```
      </xs:element>
           </xs:sequence>
         </xs:complexType>
  </xs:element>}
  <xs:element name="e_function">
        <xs:complexType>
          <xs:sequence>
     <xs:element name="e_name" type="xs:string"/>
     <xs:element name="e_function_direction_type">
       <xs:simpleType>
             <xs:restriction base="xs:string">
               <xs:enumeration value="ST"/>
               <xs:enumeration value="TS"/>
               <xs:enumeration value="FLAT"/>
             </xs:restriction>
       </xs:simpleType>
     </xs:element>
     <xs:element ref="e_function_input_entity" minOccurs="0"/>
     <xs:element ref="e_function_output_entity" minOccurs="0"/>
           </xs:sequence>
         </xs:complexType>
  </xs:element>
  <xs:element name="e_function_input_entity">
        <xs:complexType>
          <xs:sequence maxOccurs="unbounded">
     <xs:element ref="e_sdtype" minOccurs="0"/>
           </xs:sequence>
         </xs:complexType>
  </xs:element>
  <xs:element name="e_function_output_entity">
        <xs:complexType>
          <xs:sequence maxOccurs="unbounded">
     <xs:element ref="e_sdtype" minOccurs="0"/>
           </xs:sequence>
         </xs:complexType>
  </xs:element>
</xs:schema>
```

Figure 75: The E-category XML schema, page 2.

Based on the E-model schema, let us model the XML case in Figure 76 using the E-model XML schema. It is a parsed sentence using Link Grammar [87] that returns the output in a graph and also in a constituent tree form. In the graph, two objects (a word or a clause) have a limited relation.

```
    +--------------------Xp-------------------+
    |              +---------Ost--------+     |
    |              |   +-------Ds-------+     |
    +---Wd---+-Ss*b+   |     +----AN----+     |
    |        |     |   |     |          |     |
LEFT-WALL this.p is.v an example.n sentence.n .

Constituent tree:

(S (NP This)
   (VP is
       (NP an example sentence))
   .)
```

Figure 76: Sentence parsing results in constituent tree.

Figure 77 shows the modeled E-model schema in which three selected link types are modeled.

```
<?xml version="1.0" encoding="UTF-8"?>
  <efim_category xsi:noNamespaceSchemaLocation="EFIM_Category.xsd">
         <efim_symbol>sentence</efim_symbol>
         <efim_sdtype>
                <efim_symbol>value</efim_symbol>
                <efim_rawdatatype>text</efim_rawdatatype>
         </efim_sdtype>
         <efim_category>
                <efim_symbol>linkgrammar</efim_symbol>
                <efim_sdtype>
                   <efim_symbol>constituent_parse</efim_symbol>
                   <efim_rawdatatype>text</efim_rawdatatype>
                </efim_sdtype>
                <efim_sdtype>
                   <efim_symbol>violation</efim_symbol>
                   <efim_rawdatatype>text</efim_rawdatatype>
                </efim_sdtype>
         </efim_category>
```

Figure 77: The E-category XML schema instance example, page 1.

```
      <efim_category>
         <efim_symbol>words</efim_symbol>
         <efim_function>
            <efim_symbol>LinkType_O</efim_symbol>
            <efim_function_direction_type>FLAT</efim_function_direction_type>
         </efim_function>
         <efim_function>
            <efim_symbol>LinkType_P</efim_symbol>
            <efim_function_direction_type>FLAT</efim_function_direction_type>
         </efim_function>
         <efim_function>
            <efim_symbol>LinkType_S</efim_symbol>
            <efim_function_direction_type>FLAT</efim_function_direction_type>
         </efim_function>
         <efim_function>
            <efim_symbol>LinkType_V</efim_symbol>
            <efim_function_direction_type>FLAT</efim_function_direction_type>
         </efim_function>
         <efim_category>
            <efim_symbol>word</efim_symbol>
            <efim_sdtype>
               <efim_symbol>id</efim_symbol>
               <efim_rawdatatype>integer</efim_rawdatatype>
            </efim_sdtype>
            <efim_sdtype>
               <efim_symbol>lemma</efim_symbol>
               <efim_rawdatatype>varchar</efim_rawdatatype>
            </efim_sdtype>
            <efim_sdtype>
               <efim_symbol>pos</efim_symbol>
               <efim_rawdatatype>varchar</efim_rawdatatype>
            </efim_sdtype>
         </efim_category>
      </efim_category>
   </efim_category>
</efim_category>
```

Figure 78: The E-category XML schema instance example, page 2.

APPENDIX B

EML GRAMMAR

EML extends SQL-89[1] to provide various features introduced in Chapter 10. This appendix includes the current version of EML grammar over several pages. In the grammar, "!" indicates a comment. "EML" specifies EML dependent extensions. Some EML functions need subqueries to perform iterative graph walks. Accordingly, several SQL command statements are changed to allow subqueries. Readers can test arbitrary EML statements (including the examples in Chapter 10.) using Gold Parser Builder[2].

[1] http://www.devincook.com/goldparser/grammars/index.htm
[2] http://www.devincook.com/goldparser/builder/index.htm

```
! -------------------------------------------------------------------------
! E-model language (EML)
!
! An E-model prototype is built on top of relational database.
! Its full functions are implemented using hundreds of SQL procedures
! and functions. The idea is when a database system adds the E-model
! database to existing databases, then a user can query the database
! in the mixture of structured queries in addition to E-model queries.
! EML implements such ideas by extending the SQL language to support
! both structured and unstructured queries with various feature additions.
!
! Update:
!   03/29/2009 Preliminary EML design
!   06/05/2009 Add chain expression in Id list
!
! Note: This is a preliminary version of the EML based on the SQL 89 grammar
! at http://www.devincook.com/goldparser/grammars/index.htm.
! Note, 05/02/2009: Add supports to _parent, _child, _parent[%d], _child[%d]
! Note, 05/02/2009: Add supports to Id without child specification in WHEN clause
! Note, 05/02/2009: Add supports functions like second, year as the type specifier
!                   ex) WHEN ABS(t3 - t4) <= 30 SECOND;
! -------------------------------------------------------------------------------
"Name"          = 'E-model language (EML)'
"Author"        = 'Pilho Kim'
"Version"       = '05/02/2009'
"About"         = 'EML is an extended SQL language developed on top of E-model.'

"Start Symbol" = <Query>

! ===============================================================================
! Comments
! ===============================================================================

Comment Start = '/*'
Comment End   = '*/'
Comment Line  = '--'

! ===============================================================================
! Terminals
! ===============================================================================

{String Ch 1}      = {Printable} - ["]
{String Ch 2}      = {Printable} - ['']
{Id Ch Standard}   = {Alphanumeric} + [_] + ['*']
{Id Ch Extended}   = {Printable} - ['['] - [']']

StringLiteral   = '"'{String Ch 1}*'"' | ''{String Ch 2}*''
IntegerLiteral  = {Digit}+
RealLiteral     = {Digit}+'.'{Digit}+
```

Figure 79: EML Grammar, page 1.

```
! ============================================================================
! EML namespace
! ============================================================================

! Extension for EML, allow * to specify whole database
{EML Id Ch}   = {Letter} + ['_'] + ['*']

!----- Identifiers in SQL are very complex.
Id  = ({EML Id Ch}{Id Ch Standard}* | '['{Id Ch Extended}+']')
      ('.'({EML Id Ch}{Id Ch Standard}* | '['{Id Ch Extended}+']'))*

! ============================================================================
! Rules
! ============================================================================

! Support EndofStatement ';'
<Query>       ::= <Alter Stm> <EndofStatement>
                | <Create Stm> <EndofStatement>
                | <Delete Stm> <EndofStatement>
                | <Drop Stm> <EndofStatement>
                | <Insert Stm> <EndofStatement>
                | <Select Stm> <EndofStatement>
                | <Update Stm> <EndofStatement>

! ============================================================================
! Table modification statements
! ============================================================================

! Support EndofStatement ';'
<EndofStatement> ::= ';'
                   |

! Add * to table specification
<Alter Stm>   ::= ALTER TABLE Id ADD  COLUMN <Field Def List> <Constraint Opt>
                | ALTER TABLE Id ADD  <Constraint>
                | ALTER TABLE Id DROP COLUMN Id
                | ALTER TABLE Id DROP CONSTRAINT Id
                | ALTER Id DROP COLUMN Id
```

Figure 80: EML Grammar, page 2.

```
<Create Stm>  ::= CREATE <Unique> INDEX IntegerLiteral ON Id '(' <Order List> ')'
                  <With Clause>
                | CREATE TABLE Id '(' <ID List> ')' <Constraint Opt>

<Unique>      ::= UNIQUE
                |

<With Clause> ::= WITH PRIMARY
                | WITH DISALLOW NULL
                | WITH IGNORE NULL
                |

<Field Def>   ::= Id <Type> NOT NULL
                | Id <Type>

<Field Def List> ::= <Field Def> ',' <Field Def List>
                   | <Field Def>

<Type>  ::=  BIT
           |  DATE
           |  TIME
           |  TIMESTAMP
           |  DECIMAL
           |  REAL
           |  FLOAT
           |  SMALLINT
           |  INTEGER
           |  INTERVAL
           |  CHARACTER

<Constraint Opt>  ::= <Constraint>
                    |

<Constraint>      ::= CONSTRAINT Id <Constraint Type>
                    | CONSTRAINT Id

<Constraint Type> ::= PRIMARY KEY '(' <Id List> ')'
                    | UNIQUE      '(' <Id List> ')'
                    | NOT NULL    '(' <Id List> ')'
                    | FOREIGN KEY '(' <Id List> ')' REFERENCES Id '(' <Id List> ')'

<Drop Stm>        ::= DROP TABLE Id
                    | DROP INDEX Id ON Id
```

Figure 81: EML Grammar, page 3.

```
! ============================================================================
! Update database contents
! ============================================================================

<Insert Stm>    ::= INSERT INTO Id '(' <Id List> ')' <Select Stm>
                  | INSERT INTO Id '(' <Id List> ')' VALUES '(' <Expr List> ')'

<Update Stm>    ::= UPDATE Id SET <Assign List> <Where Clause> <When Clause>

<Assign List>   ::= Id '=' <Expression> ',' <Assign List>
                  | Id '=' <Expression>

<Delete Stm>    ::= DELETE FROM Id <Where Clause> <When Clause>

! ============================================================================
! Select Statement
! ============================================================================

! Add LIMIT
<Select Stm>    ::= SELECT <Columns> <Into Clause> <From Clause> <Where Clause>
<When Clause> <Group Clause> <Having Clause> <Order Clause> <Limit Clause>

! Modify to allow adding columns after *
<Columns>       ::= <Restriction> '*'
                  | <Restriction> '*' ',' <Column List>
                  | <Restriction> <Column List>

<Column List>   ::= <Column Item> ',' <Column List>
                  | <Column Item>

! Support AS
<Column Item>   ::= <Column Source>
                  | <Column Source> AS Id      !ALIAS
                  | <Column Source> Id      !ALIAS

! Add EML path query
<Column Source> ::= <Aggregate>
                  | <EML Pathquery>
                  | Id

<Restriction>   ::= ALL
                  | DISTINCT
                  |

! Add ranking order
<Aggregate>     ::= Count  '(' '*' ')'
                  | Count  '(' <Expression> ')'
                  | Avg    '(' <Expression> ')'
                  | Min    '(' <Expression> ')'
                  | Max    '(' <Expression> ')'
                  | StDev  '(' <Expression> ')'
                  | StDevP '(' <Expression> ')'
                  | Sum    '(' <Expression> ')'
                  | Var    '(' <Expression> ')'
                  | VarP   '(' <Expression> ')'
```

Figure 82: EML Grammar, page 4.

```
! Add EML path query
<EML Pathquery> ::= _R '(' <EML Pathelement> ',' <EML Pathelement> ','
                    <EML Pathelement> ')'
 | _R '(' <EML Pathquery> ',' <EML Pathelement> ',' <EML Pathelement> ')'
 | _R '(' <EML Pathelement> ',' <EML Pathelement> ',' <EML Pathquery> ')'
 | _R '(' <Column Item> ',' <EML Pathelement> ',' <EML Pathquery> ')'
 | _R '(' <Column Item> ',' <EML Pathelement> ',' <EML Pathelement> ')'
 |

<EML Pathelement> ::= StringLiteral
                | StringLiteral '/' <EML Pathelement>
                | '%' IntegerLiteral
                | '*'

! Add semantic functions
<EML Semantic Search>     ::= _ANTPTR                 '(' <Expression> ')'
                            | _SYNS                   '(' <Expression> ')'
                            | _SEEALSO                '(' <Expression> ')'
                            | _ATTRIBUTE              '(' <Expression> ')'
                            | _VERBGROUP              '(' <Expression> ')'
                            | _PPLPTR                 '(' <Expression> ')'
                            | _PERTPTR                '(' <Expression> ')'
                            | _DERIVATION             '(' <Expression> ')'
                            | _CLASSIFICATION         '(' <Expression> ')'
                            | _CLASS                  '(' <Expression> ')'
                            | _CLASSIF_CATEGORY       '(' <Expression> ')'
                            | _CLASSIF_USAGE          '(' <Expression> ')'
                            | _CLASSIF_USAGEDRN       '(' <Expression> ')'
                            | _CLASS_CATEGORY         '(' <Expression> ')'
                            | _CLASS_USAGE            '(' <Expression> ')'
                            | _CLASS_USAGEDRN         '(' <Expression> ')'
                            | _HYPERPTR               '(' <Expression> ')'
                            | _HYPOPTR                '(' <Expression> ')'
                            | _INSTANCE               '(' <Expression> ')'
                            | _PHN                    '(' <Expression> ')'
                            | _HASPARTPTR             '(' <Expression> ')'
                            | _HASMEMBERPTR           '(' <Expression> ')'
                            | _ISMEMBERPTR            '(' <Expression> ')'
                            | _ISSTUFFPTR             '(' <Expression> ')'


<Into Clause>   ::= INTO Id
                  |

<From Clause>   ::= FROM <Id List> <Join Chain>

<Join Chain>    ::= <Join> <Join Chain>
                  |
```

Figure 83: EML Grammar, page 5.

```
<Join>          ::= INNER JOIN <Id List> ON Id '=' Id
                  | LEFT  JOIN <Id List> ON Id '=' Id
                  | RIGHT JOIN <Id List> ON Id '=' Id
                  |       JOIN <Id List> ON Id '=' Id


<Where Clause>  ::= WHERE <Expression>
                  |

<When Clause>   ::= WHEN <Expression>
                  |

<Group Clause>  ::= GROUP BY <Id List>
                  |

<Order Clause>  ::= ORDER BY <Order List>
                  |

! Change to include expression
<Order List>    ::= <Expression> <Order Type> ',' <Order List>
                  | <Expression> <Order Type>

<Order Type>    ::= ASC
                  | DESC
                  |

<Having Clause> ::= HAVING <Expression>
                  |

<Limit Clause> ::= LIMIT IntegerLiteral
                 |
! ================================================================
! Expressions
! ================================================================

<Expression>  ::= <And Exp> OR <Expression>
                | <And Exp>

<And Exp>     ::= <Not Exp> AND <And Exp>
                | <Not Exp>

<Not Exp>     ::= NOT <Pred Exp>
                | <Pred Exp>
```

Figure 84: EML Grammar, page 6.

```
! Add REGEXP
! Extend expression to include the range selection
<Pred Exp>    ::= <Add Exp> BETWEEN <Add Exp> AND <Add Exp>
                | <Add Exp> NOT BETWEEN <Add Exp> AND <Add Exp>
                | <Value> IS NOT NULL
                | <Value> IS NULL
                | <Add Exp> REGEXP StringLiteral
                | <Add Exp> LIKE StringLiteral
                | <Add Exp> LIKE <EML Semantic Search>
                | <Add Exp> IN   <Tuple>
                | <Add Exp> '='  <Add Exp>
                | <Add Exp> '<>' <Add Exp>
                | <Add Exp> '!=' <Add Exp>
                | <Add Exp> '>'  <Add Exp>
                | <Add Exp> '>=' <Add Exp>
                | <Add Exp> '<'  <Add Exp>
                | <Add Exp> '<'  <Add Exp> '<' <Add Exp>
                | <Add Exp> '<='  <Add Exp> '<' <Add Exp>
                | <Add Exp> '<'  <Add Exp> '<=' <Add Exp>
                | <Add Exp> '<=' <Add Exp>
                | <Add Exp>

<Add Exp>     ::= <Add Exp> '+' <Mult Exp>
                | <Add Exp> '-' <Mult Exp>
                | <Mult Exp>

<Mult Exp>    ::= <Mult Exp> '*' <Negate Exp>
                | <Mult Exp> '/' <Negate Exp>
                | <Negate Exp>

<Negate Exp>  ::= '-' <Value>
                | <Value>

! Support function literals
! Add temporal expression
<Value>       ::= <Tuple>
                | ID
                | IntegerLiteral
                | RealLiteral
                | StringLiteral
                | <Function>
                | NULL
                | IntegerLiteral 'day'
                | IntegerLiteral 'week'
                | IntegerLiteral 'month'
                | IntegerLiteral 'year'
```

Figure 85: EML Grammar, page 7.

```
!---- Support specific function literals
<Function> ::= Now '()'
            | Second  '()' | Second  '(' <Value> ')'
            | Minute  '()' | Minute  '(' <Value> ')'
            | Hour  '()' | Hour  '(' <Value> ')'
            | Day  '()' | Day  '(' <Value> ')'
            | Week  '()' | Week  '(' <Value> ')'
            | Month  '()' | Month  '(' <Value> ')'
            | Year    '()' | Year    '(' <Value> ')'
            | Curdate '()'
            | Current_time '()'
            | Curdate_timestamp '()'
            | Curtime '()'
            | Abs '(' <Expression> ')'
            | _Popularity '(' <Value> ')'

<Tuple>       ::= '(' <Select Stm> ')'
                | '(' <Expr List> ')'

<Expr List>  ::= <Expression> ',' <Expr List>
               | <Expression>

<Id List>     ::= <Id Member> ',' <Id List>
                | <Id Member>

! <Id Member>   ::= Id
!                | Id Id

! Support 'AS' and sub query
<Id Member>   ::= Id
                | Id Id
                | Id AS Id
                | '(' <Select Stm> ')'
                | '(' <Select Stm> ')' Id
                | '(' <Select Stm> ')' AS Id
```

Figure 86: EML Grammar, page 8.

REFERENCES

[1] Abadi, M. and Cardelli, L., *A Theory of Objects*. Springer, 1996.

[2] Aho, A., Sethi, R., and Ullman, J., "Compilers: principles, techniques, and tools," *Reading, MA,*, 1986.

[3] Alexander, N., Lopez, X., Ravada, S., Stephens, S., and Wang, J., "RDF Data Model in Oracle," in *W3C Workshop on Semantic Web for Life Sciences*, 2004.

[4] Allan, J., Carterette, B., Aslam, J., Pavlu, V., Dachev, B., Kanoulas, E., and Boston, M., "Million query track 2007 overview," 2007.

[5] Allen, J. F., "Maintaining knowledge about temporal intervals," *Communications of the ACM*, vol. 26, no. 11, pp. 832–843, 1983.

[6] Amer-Yahia, S., Case, P., Rölleke, T., Shanmugasundaram, J., and Weikum, G., "Report on the DB/IR panel at SIGMOD 2005," *ACM SIGMOD Record*, vol. 34, no. 4, pp. 71–74, 2005.

[7] Angles, R. and Gutierrez, C., "Survey of graph database models," *ACM Comput. Surv.*, vol. 40, no. 1, pp. 1–39, 2008.

[8] Bachelard, G., *The Poetics of Space*. Beacon Press, 1994.

[9] Bachelard, G. and Chimisso, C., *Dialectic of Duration*. Clinamen Press, 2000.

[10] Bachman, C., "The programmer as navigator," *Communications of the ACM*, vol. 16, no. 11, pp. 653–658, 1973.

[11] Balakrishnan, R. and Ranganathan, K., *A Textbook of Graph Theory*. Springer, 2000.

[12] Bayer, R. and McCreight, E., "Organization and Maintenance of Large Ordered Indices," *Acta Informatica*, vol. 1, no. 3, pp. 173–189, 1972.

[13] Beckett, D., *RDF/XML Syntax Specification*. http://www.w3.org/TR/rdf-syntax-grammar/, February 2004.

[14] Bello, R., Dias, K., Downing, A., Feenan, J., Finnerty, J., Norcott, W., Sun, H., Witkowski, A., and Ziauddin, M., "Materialized Views In Oracle," in *Proceedings of the international conference on very large data bases*, vol. 24, pp. 1998–08, INSTITUTE OF ELECTRICAL & ELECTRONICS ENGINEERS, 1998.

[15] Berglund, A., Boag, S., Chamberlin, D., Fernandez, M., Kay, M., Robie, J., and Simeon, J., "XML Path Language (XPath) 2.0," *W3C Working Draft*, vol. 15, 2002.

[16] Bergson, H., *Creative Evolution*. Mineola, 1998.

[17] Beyer, K., Cochrane, R., Josifovski, V., Kleewein, J., Lapis, G., Lohman, G., Lyle, B., Özcan, F., Pirahesh, H., Seemann, N., and others, "System RX: one part relational, one part XML," in *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pp. 347–358, ACM New York, NY, USA, 2005.

[18] Bloesch, A. and Halpin, T., "ConQuer: A Conceptual Query Language," *Conceptual Modeling-ER*, vol. 96, pp. 121–133, 1996.

[19] Boag, S., Chamberlin, D., Fernandez, M., Florescu, D., Robie, J., Simeon, J., and Stefanescu, M., "XQuery 1.0: An XML Query Language," *W3C Working Draft*, vol. 15, 2002.

[20] Bohannon, P., Freire, J., Haritsa, J., Roy, P., and Siméon, J., "LegoDB: Customizing relational storage for XML documents," in *Proceedings of the 28th international conference on Very Large Data Bases*, pp. 1091–1094, VLDB Endowment, 2002.

[21] Böhme, T. and Rahm, E., "XMach-1: A Benchmark for XML Data Management," in *Proc. of German database conference BTW*, pp. 264–273, 2001.

[22] Botafogo, R., Rivlin, E., and Shneiderman, B., "Structural analysis of hypertexts: identifying hierarchies and useful metrics," *ACM Transactions on Information Systems (TOIS)*, vol. 10, no. 2, pp. 142–180, 1992.

[23] Brin, S. and Page, L., "The anatomy of a large-scale hypertextual Web search engine," *Computer Networks and ISDN Systems*, vol. 30, no. 1-7, pp. 107–117, 1998.

[24] Budanitsky, A. and Hirst, G., "Semantic distance in wordnet: An experimental, application-oriented evaluation of five measures," in *Workshop on WordNet and Other Lexical Resources in the North American Chapter of the Association for Computational Linguistics*, (Pittsburgh, PA), 2001.

[25] Celko, J., *Joe Celko's Trees and Hierarchies in SQL for Smarties*. Elsevier, 2004.

[26] Chamberlin, D. D. and Boyce, R. F., "Sequel: A structured english query language," in *FIDET '74: Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control*, (New York, NY, USA), pp. 249–264, ACM Press, 1974.

[27] Chen, P. P.-S. S., "The entity-relationship model: Toward a unified view of data," *ACM Transactions on Database Systems*, vol. 1, pp. 9–36, Mar. 1976.

[28] Clark, J., DeRose, S., and others, "XML path language (XPath) version 1.0," *W3C recommendation*, vol. 16, p. 1999, 1999.

[29] Clarke, C., Craswell, N., and Soboroff, I., "Overview of the TREC 2004 terabyte track," in *Proceedings of the 13th Text REtrieval Conference, Gaithersburg, USA*, 2004.

[30] Codd, E. F., "A relational model of data for large shared data banks," *Commun. ACM*, vol. 13, pp. 377–387, June 1970.

[31] Codd, E. F., *Relational completeness of database sub-languages*, pp. 65–98. San Jose, California: Prentice Hall and IBM Research Report RJ 987, 1972.

[32] Codd, E. F., "Data models in database management," in *Proceedings of the 1980 workshop on Data abstraction, databases and conceptual modeling*, (New York, NY, USA), pp. 112–114, ACM Press, 1980.

[33] Codd, E., *The relational model for database management: version 2*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1990.

[34] Cook, D., *Gold parsing system*. http://www.devincook.com/goldparser/, 2004.

[35] Council, T., "TPC Benchmark H (Decision Support) Standard Specification Revision 1.3. 0 (TPC-H)," 1999.

[36] Cutting, D. and Pedersen, J., "Optimization for dynamic inverted index maintenance," in *Proceedings of the 13th annual international ACM SIGIR conference on Research and development in information retrieval*, pp. 405–411, ACM Press New York, NY, USA, 1989.

[37] de Champeaux, D., "Bidirectional Heuristic Search Again," *Journal of the ACM (JACM)*, vol. 30, no. 1, pp. 22–32, 1983.

[38] DeRose, S., Daniel Jr, R., and Maler, E., "XML pointer language (XPointer)," *World Wide Web Consortium Working Draft*, 1998.

[39] DeRose, S., Maler, E., Orchard, D., and Trafford, B., "XML linking language (XLink) version 1.0," *W3C Candidate Recommendation CR-xlink-20000703, World Wide Web Consortium (W3C), July*, 2000.

[40] Derrida, J., "A Certain Impossible Possibility of Saying the Event," *Critical Inquiry*, vol. 33, no. 2, 2007.

[41] Dragan, F. and Gardarin, G., "Benchmarking an XML Mediator," *ICEIS*, pp. 191–196, 2005.

[42] Eisenberg, A. and Melton, J., "SQL: 1999, formerly known as SQL3," *ACM SIGMOD Record*, vol. 28, no. 1, pp. 131–138, 1999.

[43] Eisenberg, A., Melton, J., Kulkarni, K., and Zemke, F., "SQL: 2003 has been published," *ACM SIGMOD Record*, vol. 33, no. 1, pp. 119–126, 2004.

[44] Ennser, L., Delporte, C., Oba, M., and Sunil, K., *Integrating XML with DB2 XML Extender and DB2 Text Extender*. IBM, International Technical Support Organization, 2000.

[45] Fallside, D. and others, "XML Schema Part 0: Primer," *W3C Recommendation*, vol. 2, 2001.

[46] Fellbaum, C. and Vossen, P., "Connecting the Universal to the Specific: Towards the Global Grid," *Lecture Notes in Computer Science*, vol. 4568, p. 1, 2007.

[47] Fellbaum, C. D., *WordNet: An Electronic Lexical Database*. MIT Press, 1998.

[48] Gemis, M., Paredaens, J., Thyssens, I., and Van den Bussche, J., "GOOD: a graph-oriented object database system," *ACM SIGMOD Record*, vol. 22, no. 2, pp. 505–510, 1993.

[49] Gilarranz, J., Gonzalo, J., and Verdejo, F., "An Approach to Conceptual Text Retrieval Using the EuroWordNet Multilingual Semantic Database," *Electronic Working Notes of the AAAI Spring Symposium on Cross-Language Text and Speech Retrieval*, 1997.

[50] Goguen, J. and Roşu, G., "Institution Morphisms," *Formal Aspects of Computing*, vol. 13, no. 3, pp. 274–307, 2002.

[51] Goguen, J., "Data, schema, ontology, and logic integration," *Logic Journal of the IGPL*, vol. 13, no. 6, 2006.

[52] Goodrich, M. and Tamassia, R., *Algorithm design*. Wiley New York, 2002.

[53] Gou, G. and Chirkova, R., "Efficiently querying large XML data repositories: a survey," *IEEE Transactions on Knowledge and Data Engineering*, vol. 19, no. 10, pp. 1381–1403, 2007.

[54] Gray, J., "The Transaction Concept: Virtues and Limitations," *Proceedings of the Very Large Database conference*, pp. 144–154, 1981.

[55] Gruber, T., "A translation approach to portable ontology specifications," *Knowledge Acquisition*, vol. 5, pp. 199–199, 1993.

[56] Gupta, A., Liu, B., Kim, P., and Jain, R., "Using stream semantics for continuous queries in media stream processors," *ICDE Demo*, p. 854, 2004.

[57] Guting, R., "GraphDB: Modeling and Querying Graphs in Databases," *Proceedings of the International conference on Very Large Data Bases*, pp. 297–308, 1994.

[58] Guven, S., Podlaseck, M., and Pingali, G., "PICASSO: Pervasive Information Chronicling, Access, Search, and Sharing for Organizations," *Pervasive Computing and Communications, 2005. PerCom 2005. Third IEEE International conference on*, pp. 341–350, 2005.

[59] Gyöngyi, Z., Garcia-Molina, H., and Pedersen, J., "Combating web spam with trustrank," in *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, pp. 576–587, VLDB Endowment, 2004.

[60] Gyssens, M., Paredaens, J., van den Bussche, J., and van Gucht, D., "A graph-oriented object database model," *IEEE Transactions on Knowledge and Data Engineering*, vol. 6, no. 4, pp. 572–586, 1994.

[61] Gyssens, M., Paredaens, J., and Van Gucht, D., "A graph-oriented object model for database end-user interfaces," *ACM SIGMOD Record*, vol. 19, no. 2, pp. 24–33, 1990.

[62] Halpin, T., "Object-Role Modeling (ORM/NIAM)," *Handbook on Architectures of Information Systems*, pp. 81–101, 1998.

[63] Halpin, T., "ORM 2," *Lecture notes in computer science*, vol. 3762, p. 676, 2005.

[64] Halverson, A., Josifovski, V., and Lohman, G., "Rox: Relational over xml," in *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, pp. 264–275, VLDB Endowment, 2004.

[65] Harman, D., Baeza-Yates, R., Fox, E., and Lee, W., *Inverted files*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1992.

[66] Hendler, J. and McGuinness, D., "The DARPA Agent Markup Language," *IEEE Intelligent Systems*, vol. 15, no. 6, pp. 67–73, 2000.

[67] Hidders, J., "Typing Graph-Manipulation Operations," *Proceedings of the 9th International conference on Database Theory*, pp. 394–409, 2003.

[68] Higham, N., *Handbook of Writing for the Mathematical Sciences*. Society for Industrial Mathematics, 1998.

[69] HULL, R. and KING, R., "Semantic Database Modeling: Survey, Applications, and Research Issues," *ACM Computing Surveys*, vol. 19, no. 3, 1987.

[70] Huth, M. and Ryan, M., *Logic in computer science*. Cambridge University Press New York, 2000.

[71] Jain, R. and Kim, P., "Experiential meeting system," in *ACM SIGMM workshop on Experiential telepresence*, (Berkeley, California, USA), pp. 1–12, 2003.

[72] Jensen, C., Dyreson, C., Böhlen, M., Clifford, J., Elmasri, R., Gadia, S., Ggrandi, F., Hayes, P., Jajodia, S., Käfer, W., and others, "The consensus glossary of temporal database concepts: February 1998 version," *Lecture notes in computer science*, pp. 367–405, 1998.

[73] Kent, W., "A simple guide to five normal forms in relational database theory," *Communications of the ACM*, vol. 26, no. 2, pp. 120–125, 1983.

[74] Kim, J., *Supervenience and Mind: selected philosophical essays*. Cambridge University Press, 1993.

[75] Kim, P., Gargi, U., and Jain, R., "Event-based multimedia chronicling systems," in *CARPE'05: Proceedings of the the 2nd ACM workshop on capture, archival and retrieval of personal experiences*, (Singapore), pp. 1–12, 2005.

[76] Kim, P. and Jain, R., "Heterogeneous media events processing systems," in *ACM SIGMM Workshop on Effective Telepresence*, (New York, NY, USA), pp. 52–54, 2004.

[77] Kim, P. and Jain, R., "Category-based functional information modeling for echronicles," in *IEEE workshop on electronic chronicles*, (Atlanta, GA, USA), pp. 17–24, 2006.

[78] Kim, P. and Madisetti, V., "Hybrid graph data model implementation on the relational database system," *in preparation for submission to the IEEE Transactions on Knowledge and Data Engineering*, 2009.

[79] Kim, P., Podlaseck, M., and Pingali, G., "Personal chronicling tools for enhancing information archival and collaboration in enterprises," in *1st ACM Workshop on Continuous Archival and Retrieval of Personal Experiences*, (New York, New York, USA), pp. 55–65, 2004.

[80] Kimball, R. and Ross, M., *The data warehouse toolkit*. John Wiley & Sons New York, 1996.

[81] Knuth, D., "backus normal form vs. Backus Naur form," *Communications of the ACM*, vol. 7, no. 12, pp. 735–736, 1964.

[82] Knuth, D., *The Art of Computer Programming*. Addison Wesley Publishing Company, 1973.

[83] Korf, R., "Real-time heuristic search: New results," in *Proceedings of the Seventh National Conference on Artificial Intelligence*, vol. 1, pp. 139–144, 1988.

[84] Koubarakis, M., Sellis, T., and al., e., *Spatio-Temporal Databases: The Chorochronos Approach*. Springer, 2003.

[85] Kunii, H., "DBMS with graph data model for knowledge handling," *Proceedings of the 1987 Fall Joint Computer conference on Exploring technology: today and tomorrow table of contents*, pp. 138–142, 1987.

[86] Kuper, G. and Vardi, M., "A new approach to database logic," *Proceedings of the 3rd ACM SIGACT-SIGMOD symposium on Principles of database systems*, pp. 86–96, 1984.

[87] Lafferty, J., Sleator, D., and Temperley, D., "Grammatical trigrams: A probabilistic model of link grammar," in *AAAI conference on Probabilistic Approaches to Natural Language*, pp. 89–97, 1992.

[88] Leach, P., Mealling, M., and Salz, R., "A Universally Unique IDentifier (UUID) URN Namespace," *RFC4122, July*, 2005.

[89] Levas, A., Pingali, G., Podlaseck, M., Murdock, J., Center, I., and Hawthorne, N., "Exploiting pervasive enterprise chronicles using unstructured information management," in *Pervasive Services, 2005. ICPS'05. Proceedings. International Conference on*, pp. 239–248, 2005.

[90] Levene, M., *An Introduction to Search Engines and Web Navigation*. Addison Wesley Publishing Company, 2006.

[91] Levene, M. and Loizou, G., "A graph-based data model and its ramifications," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 7, no. 5, pp. 809–823, 1995.

[92] Levene, M. and Loizou, G., "Why is the snowflake schema a good data warehouse design?," *Information Systems*, vol. 28, no. 3, pp. 225–240, 2003.

[93] Levene, M. and Poulovassilis, A., "An object-oriented data model formalised through hypergraphs," *Data Knowledge Engineering*, vol. 6, no. 3, pp. 205–225, 1991.

[94] Li, L., Shang, Y., and Zhang, W., "Improvement of HITS-based algorithms on web documents," in *Proceedings of the 11th international conference on World Wide Web*, pp. 527–535, ACM Press New York, NY, USA, 2002.

[95] Li, Y., Debbie, S., Lee, Z., and Wadhwa, U., "X007: Applying 007 Benchmark to XML Query Processing Tool," in *Proceedings 10 th International Conference on Information and Knowledge Management*, pp. 167–174, 2001.

[96] Lientz, B., "Issues in Software Maintenance," *ACM Computing Surveys (CSUR)*, vol. 15, no. 3, pp. 271–278, 1983.

[97] Marchetti, A., Tesconi, M., Ronzano, F., Rosella, M., Bertagna, F., Monachini, M., Soria, C., Calzolari, N., Huang, C., and Hsieh, S., "Towards an Architecture for the Global-WordNet Initiative," in *Proceedings of the 3rd Italian Semantic Web Workshop Semantic Web Applications and Perspectives (SWAP 2006), Pisa, Italy*, pp. 18–20, 2006.

[98] Marcos, E., Vela, B., and Cavero, J., "A Methodological Approach for Object-Relational Database Design using UML," *Software and Systems Modeling*, vol. 2, no. 1, pp. 59–72, 2003.

[99] Marsh, J. and Orchard, D., "XML Inclusions (XInclude) Version 1.0," *W3C Candidate Recommendation, April*, 2004.

[100] McGuinness, D., van Harmelen, F., and others, "OWL Web Ontology Language Overview," *W3C Recommendation*, vol. 10, pp. 2004–03, 2004.

[101] Moro, M., Lim, L., and Chang, Y., "Schema advisor for hybrid relational-XML DBMS," *proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pp. 959–970, 2007.

[102] Nicola, M. and Van der Linden, B., "Native XML support in DB2 universal database," in *Proceedings of the 31st international conference on Very large data bases*, pp. 1164–1174, VLDB Endowment, 2005.

[103] Nijssen, G. M. and Halpin, T. A., *Conceptual Schema and Relational Database Design: A Fact Oriented Approach*. Prentice Hall, August 1989.

[104] Noy, N. F. and Musen, M. A., "Ontology versioning in an ontology management framework," *IEEE Intelligent Systems and Their Applications*, vol. 19, no. 4, pp. 6–13, 2004.

[105] OpenGISConsortium, "Opengis implementation specification for geographic information - simple feature access - part 2: Sql option," tech. rep., Open Geospatial Consortium, 1999.

[106] Oracle, "Oracle xml db vs. db2 v9.1 purexml: De-fanging viper," 2006.

[107] Özcan, F., Chamberlin, D., Kulkarni, K., and Michels, J., "Integration of SQL and XQuery in IBM DB2," *IBM Systems Journal*, vol. 45, no. 2, pp. 245–270, 2006.

[108] Paredaens, J., Peelman, P., and Tanca, L., "G-Log: a graph-based query language," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 7, no. 3, pp. 436–453, 1995.

[109] Patil, R., Fikes, R. F., Patel-Schneider, P. F., McKay, D., Finin, T., Gruber, T., and Neches, R., "The DARPA knowledge sharing effort: Progress report," in *KR'92. Principles of Knowledge Representation and Reasoning: proceedings of the Third International conference* (Nebel, B., Rich, C., and Swartout, W., eds.), pp. 777–788, San Mateo, California: Morgan Kaufmann, 1992.

[110] Peckham, J. and Maryanski, F., "Semantic data models," *ACM Computing Surveys (CSUR)*, vol. 20, no. 3, pp. 153–189, 1988.

[111] Pingali, G., Jain, R., Center, I., and Hawthorne, N., "Electronic chronicles: Empowering individuals, groups, and organizations," in *IEEE International Conference on Multimedia and Expo, 2005. ICME 2005*, pp. 1540–1544, 2005.

[112] Pingali, G., Tian, Y., Ebadollahi, S., Pelecanos, J., Podlaseck, M., and Stavropoulos, H., "An end-to-end eChronicling System for Mobile Human Surveillance," in *IEEE Conference on Computer Vision and Pattern Recognition, 2007. CVPR'07*, pp. 1–2, 2007.

[113] Provost, W., "Normalizing xml, part i,ii," 2002.

[114] Raeburn, I., *Graph Algebras*. American Mathematical Society, 2005.

[115] Rahm, E. and Bernstein, P., "An online bibliography on schema evolution," *ACM SIGMOD Record*, vol. 35, no. 4, pp. 30–31, 2006.

[116] Rahm, E. and Bernstein, P. A., "A survey of approaches to automatic schema matching," *The VLDB Journal*, vol. 10, no. 4, pp. 334–350, 2001.

[117] Rayward-Smith, V., Osman, I., Reeves, C., and Smith, G., *Modern heuristic search methods*. Wiley New York, 1996.

[118] Roddick, J. F., "A survey of schema versioning issues for database systems," *Information and Software Technology*, vol. 37, no. 7, pp. 383–393, 1995.

[119] Rundensteiner, E., Kuno, H., and Zhou, Y., "Incremental maintenance of materialized path query views," in *Proc. OOIS, Intąŕl Conf. Object-Oriented Information Systems*, 1998.

[120] Schmidt, A., Waas, F., Kersten, M., Carey, M., Manolescu, I., and Busse, R., "XMark: a benchmark for XML data management," in *Proceedings of the 28th international conference on Very Large Data Bases-Volume 28*, pp. 974–985, VLDB Endowment, 2002.

[121] Shanmugasundaram, J., Kiernan, J., Shekita, E., Fan, C., and Funderburk, J., "Querying XML Views of Relational Data," in *Proceedings of the 27th International Conference on Very Large Data Bases*, pp. 261–270, Morgan Kaufmann Publishers Inc. San Francisco, CA, USA, 2001.

[122] Signore, R., Stegman, M., and Creamer, J., *The ODBC Solution: Open Database Connectivity in Distributed Environments*. McGraw-Hill, Inc. New York, NY, USA, 1995.

[123] Singh, R., Li, Z., Kim, P., Pack, D., and Jain, R., "Event-based modeling and processing of digital media," in *CVDB*, pp. 19–26, 2004.

[124] Sint, L. and de Champeaux, D., "An Improved Bidirectional Heuristic Search Algorithm," *Journal of the ACM*, vol. 24, no. 2, pp. 177–191, 1977.

[125] Sint, L. and de Champeaux, D., "An improved bidirectional heuristic search algorithm," *J. ACM*, vol. 24, no. 2, pp. 177–191, 1977.

[126] Sipser, M., "Introduction to the Theory of Computation," *ACM SIGACT News*, vol. 27, no. 1, pp. 27–29, 1996.

[127] Smith, M. K., Welty, C., and Mcguinness, D. L., *OWL Web Ontology Language Guide*. http://www.w3.org/TR/2004/REC-owl-guide-20040210/, February 2004.

[128] Stephens, S. M., Rung, J., and Lopez, X., "X.: Graph data representation in oracle database 10g: Case studies in life science," *IEEE Data Eng. Bull*, vol. 27, pp. 61–67, 2004.

[129] Stonebraker, M. and Moore, D., *Object Relational DBMSs: The Next Great Wave*. Morgan Kaufmann Publishers Inc. San Francisco, CA, USA, 1995.

[130] Thalheim, B., *Entity-Relationship Modeling: Foundations of Database Technology*. Springer, 2000.

[131] Thompson, H., Beech, D., Maloney, M., Mendelsohn, N., and others, "XML Schema Part 1: Structures," *W3C Recommendation*, vol. 2, 2001.

[132] Tsai, Y., Kim, P., and Wang, Z., "A generalized image detection model for developing a sign inventory," *ASCE Journal of Computing in Civil Engineering, accepted for publication)*, 2008.

[133] van Assem, M., Gangemi, A., and Schreiber, G., "Rdf/owl representation of wordnet. W3C Working Draft 19 June 2006, 2006."

[134] Voorhees, E., "Overview of TREC 2006," in *TREC-2006, Proceedings of the Fourteenth Text Retrieval Conference. Washington, DC: Government Printing Office*, 2006.

[135] Westermann, U. and Jain, R., "Events in multimedia electronic chronicles (e-chronicles)," *International Journal on Semantic Web & Information Systems*, vol. 2, no. 2, pp. 1–23, 2006.

[136] Yuwono, B. and Lee, D., "Search and Ranking Algorithms for Locating Resources on the World Wide Web," in *Proceedings of The International Conference on Data Engineering*, pp. 164–171, Institute of Electrical and Electronics Engineering, 1996.

[137] Zobel, J. and Moffat, A., "Inverted files for text search engines," *ACM Computing Surveys (CSUR)*, vol. 38, no. 2, 2006.